

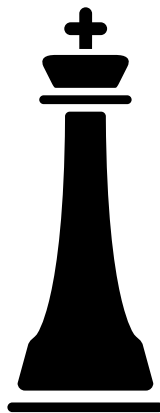


---

# CHESS ON AN INFINITE PLANE

---

A-LEVEL COMPUTER SCIENCE COURSEWORK  
H446



Andrew Keown | 1596  
Altrincham Grammar School for Boys

# Section 0: Contents

## Contents

Title Page	1
<b>Section 0: Contents</b>	
Contents	2-4
<b>Section 1: Analysis</b>	
Problem Identification	5
Problem Computability	6
Stakeholder Outline	7
Stakeholder Preferences	7-8
Existing Solutions	9-10
Survey Design	10-12
Survey Result Analysis	13-17
Client Interview	18
Interview Analysis	19
Development	19
Essential Features	19-20
Solution Limitations	20-21
Software Requirements	21-22
Hardware Requirements	22
<b>Section 2: Design</b>	
Decomposition	23-25
Basic Classes	25-28
Program Logic	28-29
AI Logic	29-32
Window Design	32
Game Design	33-34
Testing Overview	34
Testing Details	35
<b>Section 3: Development</b>	
Project Creation	36
Creating the Board	37-40
Adding Board Functionality	41-45
The Infinite Board	46-52
The Piece Class	53-58
Calculating Piece Movement	58-72
Piece Movement Tests	73-78
Turn Framework and Game States	79-81
Capturing Pieces	81-87
Check and Checkmate	87-97
Updated Piece Movement Tests	98-101
Move History	102





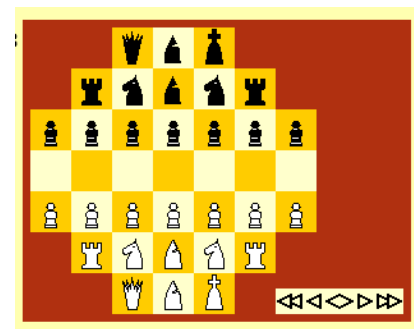
## Section 1: Analysis

### Problem Identification

Chess has been around for hundreds of years. The idea of a machine that could play chess has been around since the 18<sup>th</sup> century, and it is therefore unsurprising that computer scientists have been interested in using a computer to play chess ever since the computer was invented. A very important paper on the matter, “Programming a Computer for Playing Chess” was published in 1950 by Claude Shannon, which kickstarted the process of modern computers being programmed to play chess.

There are countless variations of chess, many of which would not be possible to play (conveniently or at all) without the use of computers. Many of these variants are easily able to be found and played online, with some simply changing the layout of pieces, others having completely different pieces or even board shapes.

The problem is that the sites I found to play these variations are all rather dated, and the games are Java applets. Google Chrome, the most widely used browser, doesn’t even support this anymore, so I had to open internet explorer, install Java, then I had to add this site to a security exception list just to run a chess variant which is low quality and unentertaining. My client, Adnan Ahmad, wants to play a chess variant, but doesn’t want to have to go through this process for a less than satisfactory experience.



“Chess with 37 squares”

One of the sites I found listed over one hundred chess variants available to play. Some of these are simpler than others from a programming perspective. However, the one variant I have not seen online is “chess on an infinite plane”, also known as infinite chess, in which the chess board is unbounded. This is the variant I will choose to tackle; an infinite board will create more of a challenge from a development point of view, and will be very different to play compared to regular chess.

## Problem Computability

There are multiple points to consider when thinking about the computability of the solution to the client's problem:

- \* Feasibility
  - Chess on an infinite plane would be very difficult to create using a physical board and pieces.
  - There is the logistical issue of not actually being able to have a board of infinite size.
  - Either a board will have to be sufficiently large that the difference is not relevant, or some method must be used to keep track of where pieces that have moved off the board are.
- \* Convenience
  - Unorthodox pieces will be difficult to obtain physically, which may cause the game to be unplayable.
  - Having a very large board or keeping track of where every piece is simply not convenient.
  - With so many pieces and such a large board, it will be difficult to calculate where pieces can move and what effects this might have (i.e. check).
- \* Usability
  - Allowing the program to calculate where pieces can move to and if any given move is legal takes a lot of mental pressure off players.
  - Use of an AI will allow just one person to play this game, which is not possible with a physical version. This will allow someone who wants to play the game to do so even if they cannot find another person.
- \* Expense
  - Both financial and “spacial” costs are minimised with a digital approach.
  - A physical solution requires a board, a large number of pieces, and any additional equipment needed to track everything.
  - A digital solution only takes up space in secondary storage, and costs very little.

Overall, it seems more sensible to approach this digitally, as the very nature of chess on an infinite plane makes it difficult to reproduce in the real world.

## Stakeholder Outline

The stakeholders of this solution are people who are interested in chess, but want to play it in a new, reinvented way easily, such as my client Adnan. However, this could also be suited for chess beginners as a different but fun game. As a result, variable difficulty will probably be something the stakeholders would be interested in, so that anyone of any skill level is able to play.

The solution should solve the problem of not being able to conveniently play a chess variant, as this will be a standalone program, so there will be no need to try and find it online, and certainly no need to play in physical space. The offline nature of this solution can make it ideal for killing boredom in situations where there is no access to internet.

## Stakeholder Preferences

I need to collect data on the way the stakeholders think this solution should manifest. My main concern will be the preferences of my client, Adnan Ahmad, since this is catered to the problem he encountered. There are various ways I can collect the data I need, each with advantages and disadvantages:

- \* Surveys
  - These will allow me to collect information from a lot of people in a short amount of time, since I only have to write the survey once.
  - These do not allow me to get detailed answers, since most answers will be yes/no or a number.
  - This makes surveys suitable for deciding what from a predetermined list of features should be present in the program, and maybe even finding out how important features are relative to each other.
- \* Observations
  - This involves me observing a stakeholder playing either regular chess or a chess variant.
  - This will allow me to see which features are used more or less between versions of the game.
  - This is useful because there may be something that cannot be asked in the survey, but observing someone will give me information about the importance of it.
  - It also shows how stakeholders use all the features of a game together while playing, rather than using features individually.
  - However, this method is time-consuming, so it cannot be for as many stakeholders as a survey.
  - Observations will be a good way to decide on user experience features, such as visuals, animations and helpers.

- \* Interviews
  - These allow me to talk one-to-one with a stakeholder.
  - This allows me to collect a lot of information; I can ask them more open questions and get a detailed answer about their preferences.
  - However, it is very time consuming. I will only conduct an interview with my client because he is the person I am catering this to.
- \* Existing Solutions
  - This involves me investigating existing products of a similar description.
  - This will allow me to investigate well-established features of the variants of chess.
  - I can also look at online reviews of variants to gather additional information on what to include and what not to include.
  - This is quite a slow way to collect information, and doesn't give me any information on stakeholder experience.

After weighing up these options, I have decided that my main methods will be surveys and interviews. It is difficult to look at existing solutions since there aren't any for my specific idea, and similar solutions are rather limited technically and no reviews exist for them, so I will only cover them briefly. Observation has similar drawbacks, but I can at least look at how people actually play the game, and see if there's any missing features that impact their experience.

Surveys will allow me to ask many people opinions on basic questions, such as "Should feature X be included?", while a client interview will allow me to ask more in-depth questions. A survey should be completed before doing an interview so that me and my client are aware of the basic features that will be present, which will then allow my questioning to be much more specific.

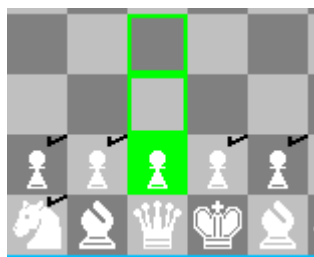
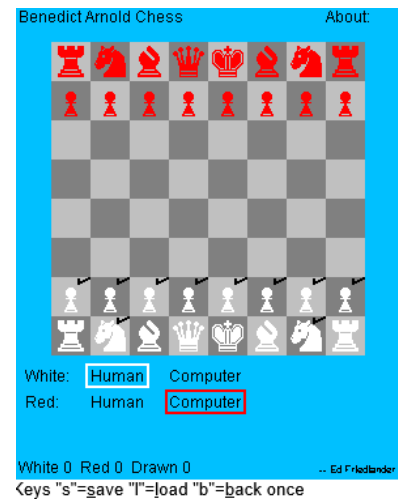


## Existing Solutions

I will find and analyse some existing chess variants to see what features they have that I should also add, and what they're missing.

The site I found for chess variants was Pathguy<sup>1</sup>, which contains over a hundred chess variants able to be played. The first thing I noticed when attempting to play one was that Google Chrome, my usual browser, does not support Java anymore. I then tried Internet Explorer, which does support Java, but the app would not run because it did not have a sufficient security rating. To fix this, I had to open the Configure Java panel, add this site to the security exception list, and then run the game. This is overall a very inconvenient process just to play chess, so I can already make an improvement over this.

Moving on, I picked a variant at random ("Benedict Arnold Chess") to look at how the game plays. The graphics are very simplistic, which was to be expected as this site is old (2005). Despite this, there are a few notable features of this game. The opponent can be either human or AI, which makes this suitable for people who don't have another person to play with. The player's possible moves are calculated every turn, and clicking on



a piece shows them. This may be desirable for some players as it means they can focus more on which move is the best move rather than what moves actually exist. Pieces that can move have a ✓ added to the top right of the square, while pieces that have no moves do not. This is also useful for the reason mentioned previously.

A perhaps even more interesting feature is the rewind, save and load feature, seen at the bottom in the first image. 'back once' rewinds the board by one move, which means it keeps track of the last turn, but it does not go further than that. 'save' will save the current board state, and 'load' will load the last saved state. These are noteworthy because they can be useful for less experienced players, allowing them to try a move and see the outcome without fully committing to it.

These features represent things that cannot be done with a physical solution for any chess game, so adding them as part of my solution, maybe even as toggleable options, seems like the most sensible thing to do, as they will enhance the user experience.

I notice that there are some features missing too. A lack of a variable AI difficulty makes this game more difficult than intended for beginners and trivial for good players, which removes these two groups of people from finding the game enjoyable. For this reason, I think it will be important that my solution has the ability to change difficulty. Furthermore, there are no animations at all for any piece movement, and as

mentioned earlier, the graphics are very simple. This may be undesirable for my stakeholders since modern games have much better graphics and animations compared to this. I will discuss this in the client interview to see what Adnan thinks.

## Survey Design

I will try to avoid written answers in the survey when possible, since people will want to complete it quickly and multiple-choice answers will be much faster.

### 1] *Which Operating System do you use?*

This question is important because it is not easy to make an application that will work out of the box on all operating systems. However, it is expected that most responses will be a Windows variant, since this is the most commonly used OS.

### 2] *How often do you play chess, online or offline (1 being seldom, 5 being very frequently)?*

This question gives me a basic insight into how much the person plays chess in any form. This will allow me to judge how important these preferences are compared to others.

### 3] *Which opponent(s) would you want in the game?*

This question will have options “AI”, “Human” or “Both”. This will allow me to see which opponent people prefer playing against, and if they would like the option for both. This is important to know because an AI will add additional time to development.

### 4] *How important are the following features to you (1 being not important at all and 5 being very important)?*

*Variable AI Difficulty*

*Move History*

*Coordinate Display*

*Main Menu*

*Custom Colours*

*Custom Start Configuration*

*Move Rewinding*

These are various features that could be in the game. This question will allow me to gain insight into which of these are more important in the view of my stakeholders. Knowing this allows me to better create interview questions for my client, and I can then see where he agrees and disagrees.

5 What are the specifications of the computer system you use?

CPU Clock Speed: \_\_\_\_ GHz      CPU Cores: \_\_\_\_ Cores

GPU Model: \_\_\_\_\_

RAM: \_\_\_\_ GB      Secondary Storage Capacity: \_\_\_\_\_ GB

Monitor Resolution: \_\_\_\_\_

It is unlikely that any of the stakeholders will be unable to run the game on their current system, as it has very low graphical intensity. However, resolution will be an important thing to consider as many people use laptops. While newer laptops are 1920x1080, some people using older laptops may have 720p monitors, so it could be necessary to accommodate for this.

I will use Google Forms to collect this data, as this is more convenient for both me and the people filling it in; I don't have to hand out physical surveys and collect them again, the results are automatically collated for me, and it's easy for others to just click the link and fill it in.

I will provide a link to everyone in my Computing class, which should provide me with a reasonable sized set of data to work with.

Below is the finished form:

How often do you play chess, online or offline? \*

	1	2	3	4	5	
Seldom	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Very Frequently

Which opponent(s) would you want in the game? \*

- ☐ Human
- ☐ AI
- ☐ Both

How important are the following features to you (1 being not important at all and 5 being very important)? \*

	1	2	3	4	5
Variable AI Difficulty	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Move History	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Coordinate Display	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Main Menu	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Custom Colours	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Custom Start Configuration	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Move Rewinding	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Which Operating System do you use? \*

- ☐ Windows 7 and later
- ☐ Windows Vista and earlier
- ☐ Mac OS X
- ☐ Linux
- ☐ Other: \_\_\_\_\_

What is the clock speed of your CPU? \*

Your answer

What is the capacity of your hard drive? \*

Your answer

What is the resolution of your monitor? \*

- ☐ 3840x2160
- ☐ 1920x1080
- ☐ 1280x1024
- ☐ 1600x900
- ☐ 1280x720
- ☐ Other: \_\_\_\_\_

How many cores does your CPU have? \*

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 6
- ☐ 8
- ☐ 10 or more
- ☐ Other: \_\_\_\_\_

What is your GPU model?

Your answer

How much RAM do you have? \*

- ☐ 1GB or less
- ☐ 2GB
- ☐ 4GB
- ☐ 6GB
- ☐ 8GB
- ☐ 12GB or more

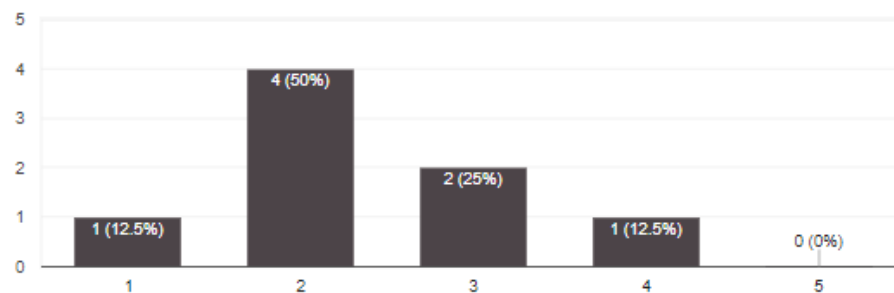
## Survey Result Analysis

### \* Question 1

- Many of the people that responded do not play chess very frequently. However, that doesn't completely invalidate their opinions on what features should be in my solution, since it is likely that most of them do understand chess at a basic level at least, and have played some online version of chess at some point.

How often do you play chess, online or offline?

8 responses

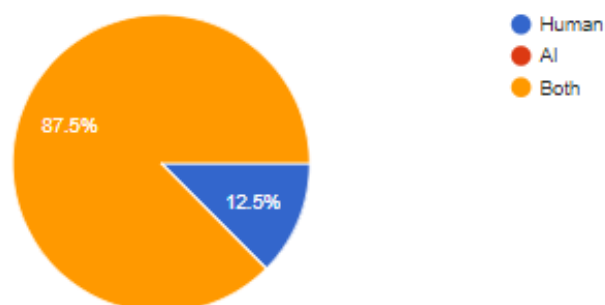


### \* Question 2

- An overwhelming majority of responses asked for both human and AI opponents in the game. This is significant because creating an AI will add to the total development time, so I must make sure I have enough time to get this feature in. However, adding support for both will increase the usefulness of my solution, since it is now both a 1-player and a 2-player game.

Which opponent(s) would you want in the game?

8 responses

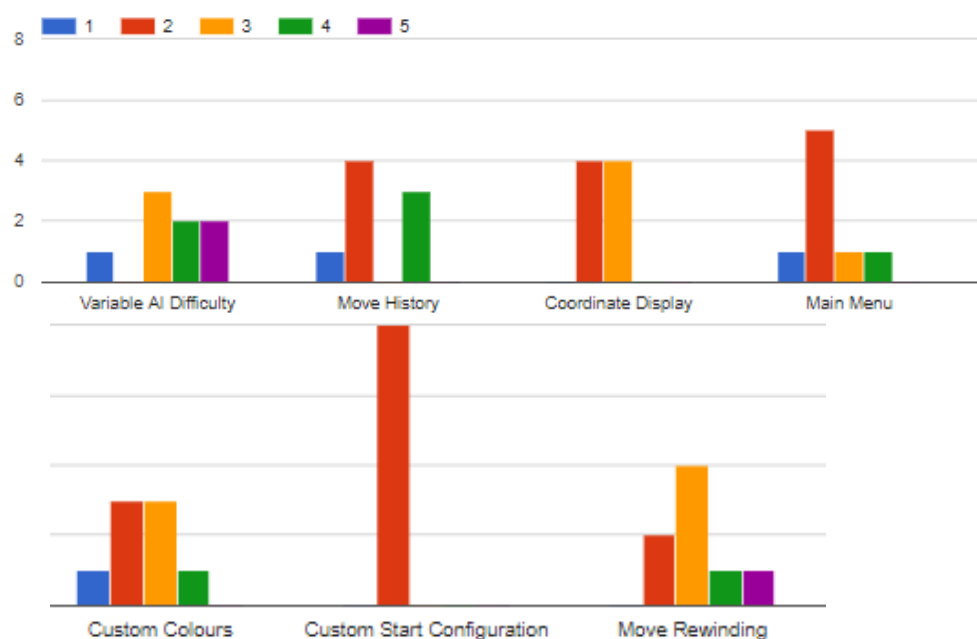


### \* Question 3

- Variable AI Difficulty seems to be considered fairly important, with an average score of 3.5. This makes sense, as a variable AI difficulty will allow the game to be played by people of all skill levels. As a result, this will be a feature I will include in the solution.

- 5 people gave Move History a 1 or 2, while 3 gave it a 4. This shows a divide between people wanting this feature and people that don't. The average score for this was 2.6, however I think it would be a useful feature, so I will need to speak to my client and see if it's something he would be interested in.
- Coordinate Display scored an average of 2.5, and it is a minor feature so it is unlikely I will put coordinates on individual squares or have a display for the current coordinates. I also haven't seen this on any existing solutions, so I will probably just go with the more common feature of displaying row and column letters/numbers around the board.
- Most people scored Main Menu a 2. I do not think a main menu is too important for a game like this; I envisage it being similar to the games that come with Windows 7, such as Solitaire or Minesweeper, in the sense that there is no main menu, and a toolbar at the top of the screen is used instead.
- Custom Colours and Start Configurations scored fairly low (2.5 and 2, respectively). This is expected because they are very minor features and don't really impact the game a huge amount. Since these are not important according to the survey results, I will not be adding them to the final solution.
- Move Rewinding scored 3.1, so it is seen as fairly important. I think it is an important feature for the same reason as Variable AI Difficulty is. I will discuss this feature with Adnan to see if it's something he thinks is important for the game.

How important are the following features to you (1 being not important at all and 5 being very important)?

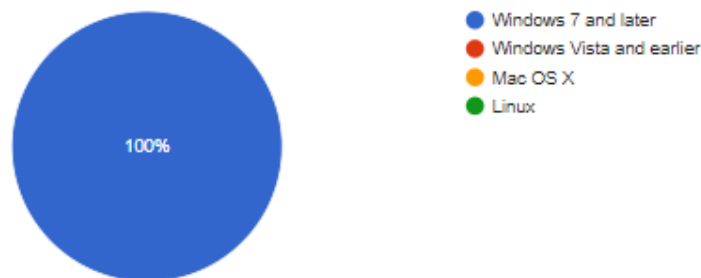


\* Question 4

- Every person who responded gave Windows 7 or later as their OS. Assuming my client also uses Windows, I will not make any attempt to support other operating systems, since it is a fair assumption that a vast majority of people that would use this software will be on a recent version of Windows. This means I am free to use my language of choice, C#, which is part of .NET.

### Which Operating System do you use?

8 responses



\* Questions 5, 6 and 7

- The people who responded to the survey have generally high quality CPUs and GPUs. The average clock speed is 3.3GHz, and 100% are 4 or more cores. My solution is very light on graphics and processing anyway so there should be no problems with performance for any modern computer system, and certainly not on any of the systems owned by the people who responded. GPU is not really a concern since the game will use the CPU because of the very low intensity graphics.

### What is the clock speed of your CPU?

8 responses

3.1
Many
3.3ghz
2.4 GHz
3.6ghz
2.9 ghz
4.1
4 GHZ

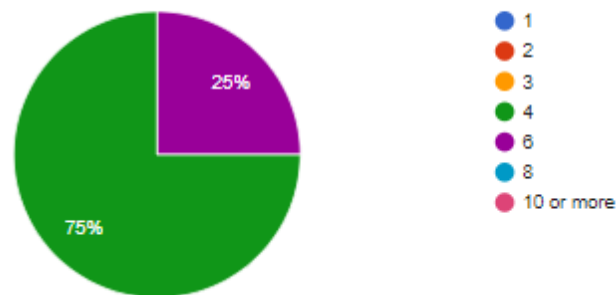
### What is your GPU model?

8 responses

GTX 1060
Gtx 1060
gtx950
Intel HD 4000
Asus 960
Gtx 950m
gtx 1080
GTX 1070

### How many cores does your CPU have?

8 responses

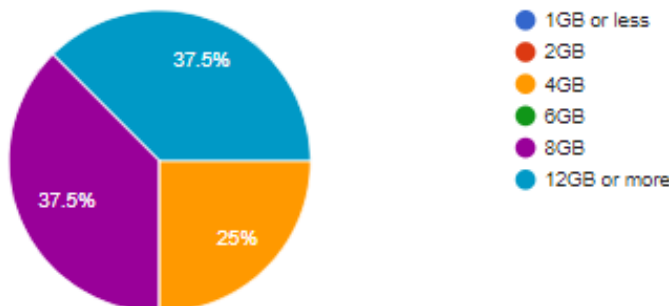


#### \* Questions 8 and 9

- Similarly, most responses have ample amounts of RAM and secondary storage space. My game will use very little of both of these, so there will certainly be no performance issues.

### How much RAM do you have?

8 responses



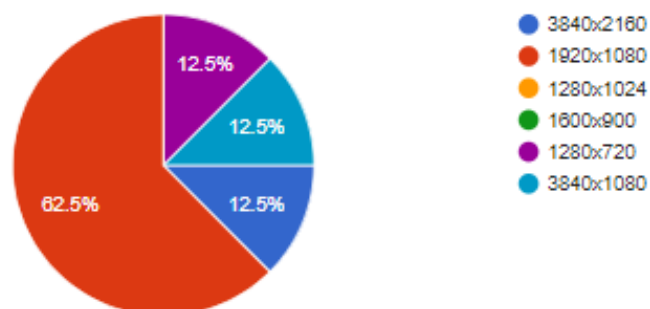
### What is the capacity of your hard drive?

8 responses

256gb
1TB HDD + 256GB SSD
1tb
1 TB
1TB HDD+ 120GB SSD
1000 gb
2Tb + 500gb ssd
2 TB

### What is the resolution of your monitor?

8 responses





\* Question 10

- 63% of responses had a 1920x1080 monitor. However, there were some responses for other sizes too, notably 1280x720. I will ask my client about the resolution of his own monitor. 1080p is the most common resolution so I must make sure the game looks correct on this resolution. 720p is used on a lot of somewhat old laptops, and I am also developing the game on a 720p monitor. This means I will need to have the game able to scale with both sizes, and test 1080p on another device. I could also add support for other resolutions, but these two are the priority.

Overall, the most important features according to this survey are those which accommodate for players of different skill levels. People were not so interested in minor customisation options. There are no problems with computer specifications in relation to possible performance issues, since my game will be lightweight and responses to the survey have good computer systems. I will discuss with my client monitor resolution to see if I need to make support for anything other than 1080p and 720p.

## Client Interview

After the surveys and existing solution research, there are still a few questions I need to clear up. This will be done with an interview with my client Adnan, as he is the person I am catering this solution to.

The things I need to decide on are those where the result from the survey is not clear and it is not obvious what my client would want without asking. These things are the main menu, colour customisation, and variable AI difficulty. I also need to ensure that my hardware requirements are suitable for my client.

*Me: Ok, so there's a few things I need to clear up before I can write the specification. First, you're using Windows 10, right?*

*Adnan: Yeah.*

*M: Ok good, and what's the resolution if your monitor?*

*A: It's 1080p.*

*M: Right, that's the technical stuff out the way. Now for actual features; what do you think about a variable AI difficulty?*

*A: I think it's a great idea, because I have found that many existing variants of chess fail to challenge me.*

*M: That's what I expected, I agree that it will be a useful feature. Is move rewinding a feature you would be interested in also?*

*A I think it's a good idea for other players, however for me, as I am looking for a chess variant with greater difficulty, I don't feel it's necessary.*

*M: A lot of people in the survey said they weren't too interested in a main menu. I was thinking of making it like those games that come with Windows, like solitaire and stuff, where there's no main menu and everything is done with a toolbar at the top. What do you think about that?*

*A: I agree, I feel it makes navigation much quicker and efficient, and means when I launch the game I can get stuck in to the game.*

*M: Ok, how about more superficial features like colours and themes. Do you want those in the game, or are they unnecessary?*

*A: I don't think this is necessary, and may be in some cases distracting to the players.*

*M: Oh, and a move history, are you interested in seeing the previous moves?*

*A: Definitely, as someone who often loses track of moves, this would really be useful to me, and I think to a lot of other players.*

*M: Ok that has cleared up those last details for me. Thanks.*

*A: No problem.*

## Interview Analysis

From the interview, we can decide on a number of features:

- The solution will be aimed solely at Windows Vista and newer version of Windows. None of the responses to the survey had older versions, and my client is on Windows 10. Visual Studio 2017 will natively support Windows Vista and after, and will support XP to an extent. To ensure that everything will always work with no compatibility issues, I will state the requirement as Windows 7 and later.
- There will not be a main menu. It scored low on the survey, and my client said he was not interested in this feature, preferring a toolbar for more efficient navigation. It is also faster for me to implement a toolbar compared to a main menu, so that is what I will do.
- There will be no customisation for colours or board configurations. These are minor features, and my client is not interested in them because they will act as distractions.

## Development

I will be developing this game using Visual Studio 2017. This is an IDE that supports many different .NET languages, such as C#, C++, Visual Basic and F#. I chose this software because it has many useful features for developers, including detailed compiler error reports, CPU and RAM usage, auto-completion, and real-time syntax error checking.

My language of choice will be C#. In Visual Studio, I can create Windows Forms with C#, which means I do not have to spend time developing my own GUI components; they are already supplied for me to use. C# is preferable for me over the other available languages which support Windows Forms because I have a reasonable amount of prior experience with the language. This means I do not need to worry about mistakes involving the use of the language (i.e. syntax errors) as much as about logical errors.

## Essential Features

Compiling the results from the existing solutions, surveys and client interview, I have decided on a list of essential features:

- \* Standalone Offline Application
  - My client's problem is that it is inconvenient to play an online chess variant because of all the steps required. I will resolve this by making my solution an offline executable in C#, so any modern Windows machine will be able to run it out of the box with no issues. This addresses the main issue of my client.
- \* Response to Resolution
  - While a large proportion of the people who completed the survey had 1080p monitors, there are some others with different resolutions such as 720p too (I am actually developing this on a laptop with a 720p

monitor). For this reason, I think it is important to account for at least these two resolutions, whether it is automatic or manual, so that the game will look correct regardless.

- \* Human and AI Opponent
  - There was a lot of request from the surveys and also from my client to have both AI and human opponents available. This is an important feature because it will allow someone to play the game even if they don't have another person to play with. This was also a feature in the chess variants researched in existing solutions.
- \* Variable AI Difficulty
  - This is seen as an important feature to both the people that filled out the survey and to my client. This gives the game a much wider audience, since inexperienced players can play without the AI being too difficult, and experienced players can challenge themselves against a difficult AI. My client has also said that this feature will allow him to continue playing the game as he improves.
- \* Move Rewinding
  - Similar to the reasons for variable difficulty; move rewinding gives less experienced players a chance to be able to play by not punishing them so hard for mistakes. More experienced players can simply not use this option to give themselves extra challenge.
- \* Move History
  - This will allow players to keep track of moves that have occurred. Due to the nature of the game, not all pieces will be visible to the player at once, so being able to find out where all pieces are will be very helpful.

## **Solution Limitations**

There are some features that will not be included because they are either not wanted or would detract too much from the development of more important features:

- \* No Online Capability
  - Online play would add another level of complexity to the game, which I do not have time to develop. This solution is not aimed at those who wish to play online; only for either one or two players.
- \* No Main Menu
  - A main menu was not seen as important by either my client or the survey responses, so it will not be implemented. A toolbar will be used instead for navigation.
- \* No Colour Customisation
  - Colour customisation is a minor feature which does not really add or detract anything in terms of actual gameplay. My client said in the interview that this would only serve as a distraction, so I will not develop this feature.

- \* Windows Vista and Later
  - Visual Studio 2017 will natively support operating systems from Vista onwards. This means I will not be supporting any previous ones, namely XP. This is unlikely to be a problem because few people are using Windows XP (none from the survey responses).

## Software Requirements

My success criteria are represented by the software requirements.

- \* Technical
  1. Generates squares on the fly\*.
  2. Game will be saved on exit\*.
  3. Can resize to fit different resolutions.
- \* Gameplay
  4. Has move rewinding.
  5. Has variable AI difficulty.
  6. Has human or AI opponent.
- \* UI
  7. Has a toolbar for navigation.
  8. Has move history on screen.
  9. Has on-screen buttons to scroll the board\*.
  10. Shows the user where they can move when they click on a piece.
  11. Shows the user which pieces have been taken\*.
- \* User Experience
  12. Has keyboard keys to scroll the board\*.
  13. Provides tutorials for unorthodox pieces.
  14. Tells user if an attempted move is invalid.

Criteria marked with \* have not been justified previously and therefore will be below:

- \* The board will begin at the size that can be seen on screen. As the user scrolls using the buttons, the board will automatically expand. This cuts down on loading time since the program does not need to load a large amount of squares at start-up, and also makes the program more memory efficient since only squares that are actually needed will be generated and saved.
- \* When the game is exited, the current game state will be saved. Some games could go on for some time depending on how the user plays, so being able to pick up a game part way through at a later time is better for user experience.
- \* Buttons will be used to scroll the board. I decided to use buttons instead of a scroll bar because I feel a scrollbar will not provide the level of precision a user would want in this game. Also there is the issue of the scroll bar becoming too small to use when the board has large dimensions. Keyboard buttons are present to allow for faster and easier scrolling if desired.
- \* Similar to the move history, a display of which pieces have been taken will be useful for the user to keep track of what has happened. Since infinite chess has

a lot of extra pieces, it can be difficult to notice if one is missing. This feature will take the responsibility of remembering which pieces are gone off the user.

## Hardware Requirements

- \* Resolution
  - 1920x1080 or 1366x768
  - 1080p is the most common monitor resolution, so it is vital that the game is compatible with this resolution. Many somewhat old laptops have a resolution of 720p (and so does the device I am developing this on), so I will also be implementing support for this.
- \* CPU
  - 1GHz, Dual Core or better
  - The game will not be very intensive at all from a computational standpoint; it is a 2D game with low resolution graphics and relatively simple algorithms. As a result, not a lot of CPU power is required.
- \* RAM
  - 1GB
  - The amount of RAM the program requires will be very small. This requirement represents the RAM you would need to run an operating system that is compatible with the game.
- \* Storage Space
  - 500MB
  - This is not a complex game overall; the only things that need to be stored aside from the program itself are images used in the game, saved games, and any user settings. Therefore a very little amount of free hard drive space is needed.
- \* GPU
  - N/A
  - Since the game uses simple 2D graphics in a Windows Form, the CPU will handle the graphics. Any CPU that is powerful enough to run the program will have an integrated GPU powerful enough to handle the graphics.

## Section 2: Design

### Decomposition

To decompose the problem, I will first consider 4 areas: Window, Game, Toolbar and Misc. Within each of these I will identify the features relevant to that area and outline the information that is needed to create each feature in the actual solution. This information is represented in the table below, with the first column being the general area, the second being a more specific feature, and the third being the information about the feature.

C# is an object-oriented language, so everything in my solution will be built around objects. I will be making full use of this by creating custom classes for various features in the game.

WINDOW	BOARD	Draw bitmap of chess board
		Contained in custom container class
		Overrides OnMouseClicked event
		Comprised of Squares (custom object) to store relative and absolute coordinates
	TOOLBAR	Displayed at top of window at all times
		Contains game settings (resolution, game restart, opponent settings) and help/tutorial
	HISTORY	Record every move and write to a text box
		Also saved to a file
		Displayed using algebraic chess notation <sup>2</sup>
	BUTTONS	Clear on game restart
		Redraw pieces to give impression of scrolling
		Pieces off the edge of the visible board are not drawn
		Board extends when scrolling at the edge of the current board
GAME	INITIALISE	Save data cleared, history cleared
		Board size is reset
		Board is drawn
		Pieces are drawn into starting configurations
	PIECES	Defined as objects with appropriate properties and methods
		Function to calculate and display available moves

		Function to move piece
	LOGIC	
		Turn system; white makes a move, and then black moves
		Only pieces of the colour of the current turn can be selected
		If AI opponent, calculate a response to the player and make a move
		If human opponent, give control to the opposite colour
		Check every turn if either king is in check or if the stalemate conditions are met
		If so, end the game
	MOVEMENT	
		Pieces show available movement when clicked
		Clicking another square will move the piece if possible
		Some situations will remove most possible moves (being in check, en passant), forcing the user to make a specific move or moves
		Cannot move King into check or checkmate
		Pieces can be moved further then the length of the board by scrolling after clicking the piece
	REWINDING	
		Will reset the board and game state to be as it was before the last move by both players
		Game state is saved to a file after every move; load data from file to achieve rewind
TOOLBAR		
	GAME	
		Start a new game; exit the game; request stalemate; forfeit game; rewind last move
	WINDOW	
		Change resolution; disable UI features (move history, movement indicator, pieces taken)
	SETTINGS	
		Change AI difficulty; change opponent; change amount buttons scroll by
	HELP	
		Tutorials for unorthodox pieces; explanation of features; general chess help
	ABOUT	
		Information about the program
MISC		
	SAVING	
		When the game is exited, game state is saved to a file and is loaded again next time the game opens



	DIFFICULTY	Difficulty controls how many moves in the future the AI will consider and the chance of the AI making the best move possible
	OPPONENT	The AI will only play if the option is selected, otherwise control will be given to the other colour piece at the end of the turn

## Basic Classes

There are a number of key classes that will be used to form the structure of the game. Each class has properties and algorithms associated with it, which will provide a certain function for the game. Due to the nature of Windows Forms, the main loop is not written by me, and only deals with the actual window. The code I will be writing to create the game is event-driven, with most functions only being called as a result of a user interaction with the window.

### *chessWin*

- \* Represents the actual window
- \* Execution of game begins in method ChessWin()
- \* Extends Form

ATTRIBUTE	TYPE	PURPOSE
board	List<Square>	A list of all the squares currently in the board
pieces	List<Piece>	A list of all pieces currently in play
size	int[]	The current dimensions of the board
bounds	int[]	Values representing where the edges of the board are
origin	int[]	The coordinates of the square 0, 0
state	GameState	The current state of the game

```

public chessWin() { //execution of game begins in this function
    state = initialising;
    InitializeComponent(); //added by the IDE, not written by me
    InitialiseBoard(); //Initialise attribute 'board'
    InitialisePieces(); //Initialise attribute 'pieces'
    drawGrid(); //draws the board and pieces after initialisation
}

public void InitialiseBoard() {
    for each square on the board graphic {
        board.Add(new Square);
    }
}

public void InitialisePieces() {
    for each piece needed for the game {
        pieces.Add(new Piece);
    }
}

public void drawGrid() {
    Graphics.Draw(bitmap of the chess board);
    foreach (Piece p in pieces) {

```

```

        Graphics.Draw(p.icon, p.x, p.y);
    }
    state = playing;
}
public void EvaluateCheck() {
    foreach (Piece in pieces) {
        if (piece = king) {
            evaluate if the king is in check or checkmate
            if so, state = blackwin or whitewin depending on which king
            display the result
        }
    }
}
}

```

## Square

- \* Represents a square on the board
- \* Used to link absolute coordinates to a square on the board

ATTRIBUTE	TYPE	PURPOSE
X	int	The X coordinate of the top left corner of the square
Y	int	The Y coordinate of the top left corner of the square
indexX	short	The X index of the square on the board
indexY	short	The Y index of the square on the board

```

public static List<Square> emptyList() {
    return new List<Square> { }; //Used to initialise lists of squares so that
    they can then be edited later
}
public override string ToString() {
    return each attribute separated by commas //Used so that we can display the
    value of a square to a label which is useful for debug
}

```

## GameContainer

- \* The control which will contain the chess board itself on the window
- \* Extends Panel

```

protected override void OnMouseMove(MouseEventArgs e) {
    //write the data of the square that the mouse is currently over to a label
    //this is useful for debugging many things throughout development
    Square cursorSquare = findSquareByCoords(e.X, e.Y)
    if (cursorSquare != null) {
        label.Text = cursorSquare.ToString();
    }
}
protected override void OnMouseClicked(MouseEventArgs e) {
    //when the user clicks a square, try and find the piece that is on that
    square, and then prepare to move the piece if applicable
    Square cursorSquare = findSquareByCoords(e.X, e.Y)
}

```

```

    Piece pieceClicked = null;
    foreach (Piece p in chessWin.pieces) {
        if (p.square == cursorSquare) { pieceClicked = p; }
    }
    if (pieceClicked != null) {
        draw movement for this piece
        state = moving;
    }
    if (state == moving) {
        move the selected piece to this square if valid
        state = playing;
    }
}

public static Square findSquareByCoords(int x, int y)
{
    //this is necessary so that we can find a square based on coordinates of
    the cursor
    foreach (Square s in chessWin.board)
    {
        if (x >= s.X && x < s.X + chessWin.sf2 && y >= s.Y && y < s.Y +
chessWin.sf2) { return s; }
    }
    return null;
}

public static Square findSquareByIndex(int indexX, int indexY)
{
    //this is necessary for any function that does something to a piece (we
    need to know where the piece is)
    foreach (Square s in chessWin.board)
    {
        if (indexX == s.indexX && indexY == s.indexY) return s;
    }
    return null;
}

```

## Piece

- \* Represents a chess piece
- \* Makes use of 2 Enumerations:
  - enum PieceType { pawn, knight, rook, bishop, queen, king, mann, hawk, chancellor, none };
  - enum PieceColour { black, white };

ATTRIBUTE	TYPE	PURPOSE
type	PieceType	The type of piece this is
square	Square	The square the piece is on
icon	Bitmap	The icon of the piece
Colour	PieceColour	The colour (black/white) of the piece

```

public Piece(PieceType t, Square s, PieceColour c) {
    //the constructor for a piece; icon is not an argument in this
    type = t; square = s; colour = c; icon = link to icon based on type
}

```

```

public List<Square> calculateMovement() {
    //calculates the available moves for the piece and returns it as a list of
    squares
    if (type == none || type == null) return Square.emptyList();
    switch (type) {
        List<Square> movement = Square.emptyList();
        calculate movement for each type
        return movement
    }
}

public static List<Piece> InitialisePieces() {
    //stores the starting configuration of the board, which is then called at
    the start of the game
    List<Piece> pieces = new List<Pieces>;
    if save data exists, load that into pieces
    else
        Pieces.Add(first piece required);
        Pieces.Add(second piece required);
        ...
    return pieces;
}

```

## Program Logic

This represents the way the core feature (the chess game) will function from a computational view (as opposed to how the AI will function or how exactly every decision is made). This is the logical progression from function to function; which function is called as a result of a given event happening.

- \* state is set to init. Initialisation happens in ChessWin(), which sets up the game for the user to begin playing, as seen above. state is set to playing.
- \* Pressing a scroll button at any time will call the event OnClick for that button. For example, pressing the scroll up button will call the following function:

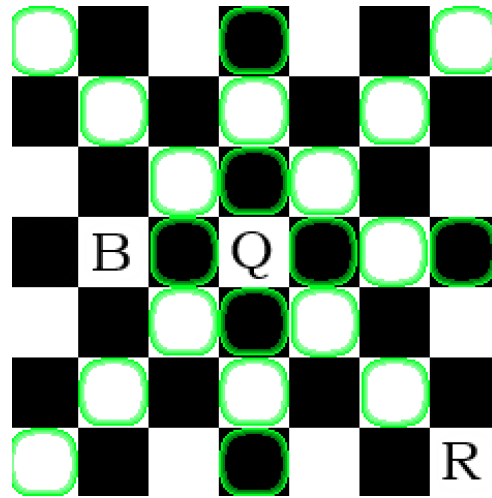
```

private void scrollUp_Click() {
    //first find the square which is in the top left corner
    Square edge = GameContainer.findSquareByCoords(0,0);
    //update the Y coordinate of each square and the origin
    0[1] += height of one square;
    foreach (square in the board) {
        square.Y += height of one square
    }
    //if the edge square has the same Y coordinate as the current upper
    Y bound, then add a new row to the board and update the bound
    if (edge.indexY == bounds[upper y]) {
        bounds[upper y] += 1;
        board.Add(new row above the old boundary);
    }
}

```

- \* Pressing somewhere on the board will call GameContainer.OnMouseClicked(), for which the algorithm can be seen earlier on. If a square with a piece is clicked on, the available moves for that piece will be drawn to the screen and state is set to moving. To the right is an example of one design for this. The queen (Q)

has been pressed by the user, which can move any number of squares orthogonally. However, 2 directions are blocked by a bishop (B) and a rook (R). This affects the available moves for the queen, which are shown using the green circles. Clicking on one of these circles will move the piece to that location using `MakeMove(Piece p, Square location)`. state is set back to playing. Call `EvaluateCheck()` to check if either king is in check or if either player has won the game.



- \* If the opponent is AI, `MakeMoveAI()` is called to calculate an appropriate (not necessarily best, depending on difficulty) move to make. This move will be made and control will return to the player. If the opponent is human, the previous step is repeated but with control of the opposite colour pieces instead (e.g. `control = PieceColour.black;`). Call `EvaluateCheck()` again.
- \* Game logic is followed throughout the process of moves being made. After every move, the board state is saved to a file in `SaveData()` and the move history is updated with `History.Update()`.
- \* One of three things will happen; a player will win, a stalemate will occur, or the game will be closed. If either player wins or a stalemate occurs, the game state is set appropriately and a message is displayed informing the player what has happened. They can then start a new game which will put the program back into initialisation. If the game is closed then the game will resume where it was left off next time it opens in `InitialisePieces()`.

## AI Logic

The first thing an AI for chess must be able to do is get a list of all moves available for every piece of one colour. This can be achieved using the function seen earlier `Piece.calculateMovement()`:

```
public List<string[]> calculateAllMovement(PieceColour c) {
    //create an empty list for the moves
    List<string[]> result = new List<string[]>();
    foreach (Piece p in pieces) {
        if (p.colour == c) {
            p.calculateMovement();
            //add each move for each piece to the list
            foreach possible move {
                result.Add(new string[] {coordinates of p, coordinates
of move});
            }
        }
    }
    return result;
}
```

The simplest AI one could make is to simply select a random move from all possible ones. This will give something that you could play against, but it will be ineffective as it will make no attempt to actually win the game.

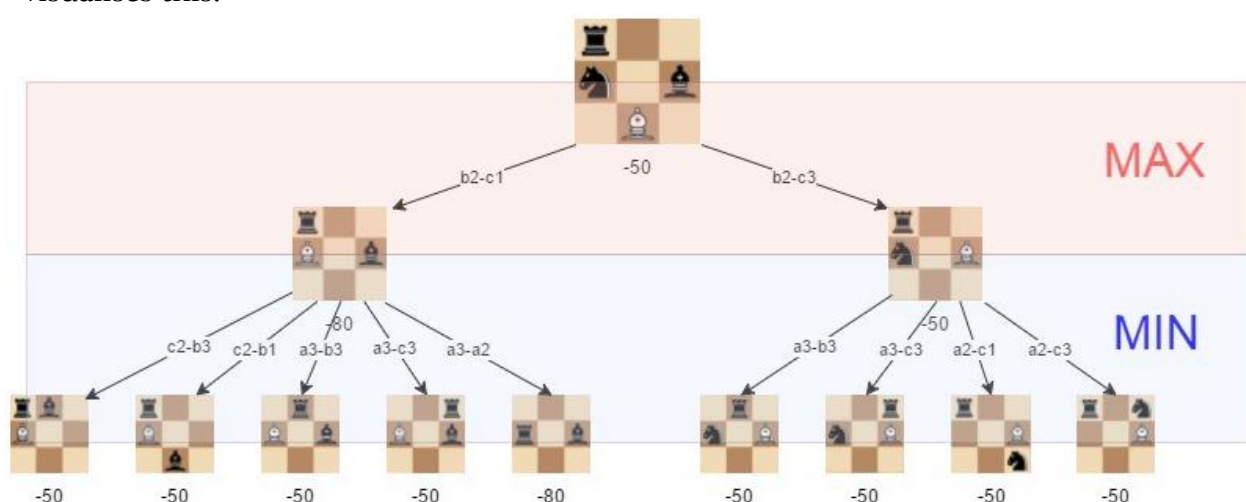
To improve this, we can decide on a metric that the AI can use to evaluate the state of the board in terms of which side is in a better position. This is the basis for many chess AIs, and is just as applicable in the infinite variant. This method works by assigning each piece of your own colour a positive value based on how useful the piece is, and each opposing piece is given a score of equal magnitude but negative. Summing the scores of every piece on the board gives a basic metric for the state of the game. An AI can then use this and select the move which gives the highest positive value. Example values for pieces are in the table below.

PIECE	SCORE
Pawn	1
Mann	2
Bishop	3
Knight	3
Hawk	4
Rook	5
Chancellor	7
Queen	9
King	50

However, all this effectively does is make the AI capture a piece if it can. Otherwise, it still just picks moves at random, which is still not an effective strategy.

The next way the AI can be improved is by searching future moves and evaluating these in the same way. Making use of the minimax algorithm will allow the AI to actively select moves that will lead to a favourable game state.

The minimax algorithm recursively searches a tree of moves and evaluates a score for each node using the same method as above. At each layer, the best move is decided by looking for either the minimum or maximum value of the child nodes. Whether we are maximising or minimising depends on the player that would be playing on the turn we are considering. Below is an image<sup>A</sup> which visualises this.



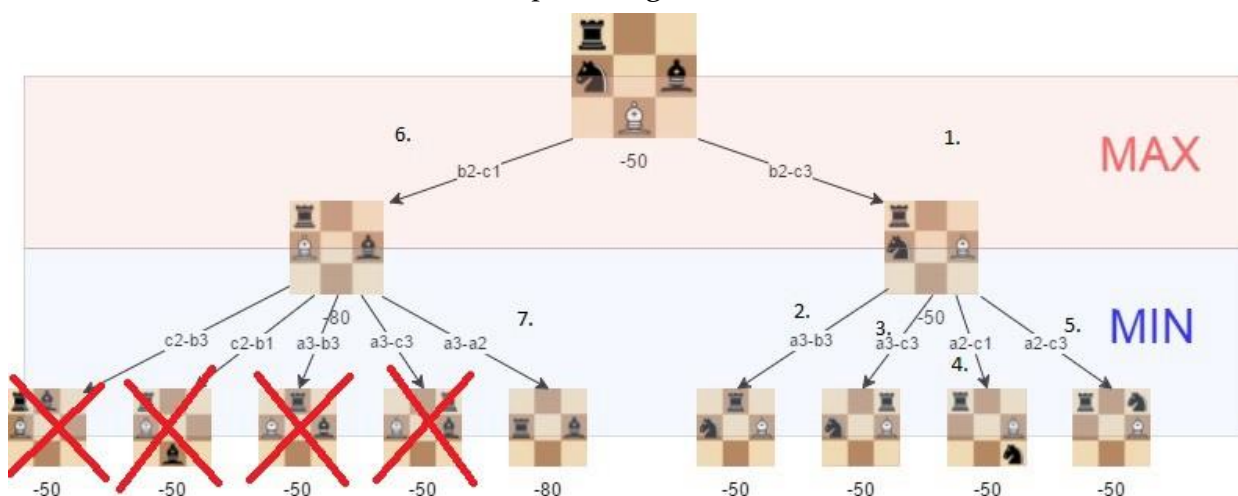
Given the 3x3 state seen at the top of the image, the minimax algorithm would return b2-c3 as the best move (the one on the right) because this will guarantee a minimum score of -50, whereas the other move could end up scoring -80. The minimum score

for white has been maximised. This algorithm will be as effective as the search depth we give it. Pseudocode for this algorithm:

```
function int maxi(int depth) {
    if (depth == 0) return evaluate();
    int max = -9999;
    for (all moves) {
        score = mini(depth - 1);
        max = math.max(score, max)
    }
    return max;
}

function int mini(int depth) {
    if (depth == 0) return -evaluate();
    int min = 9999;
    for (all moves) {
        score = maxi(depth - 1);
        min = math.min(score, min);
    }
    return min;
}
```

There are some ways this algorithm could be improved further. Alpha-beta pruning can be used to search with more depth using the same amount of resources. In the



example seen here, if the algorithm visits each node in the order they are labelled, we know that choosing b2-c3 will provide a minimum of -50. As soon as we consider the first node of the left path, we see it is -80 and therefore the minimum for this side is lower. This means we do not have to consider any of the remaining nodes on the left side, thus saving computation time.

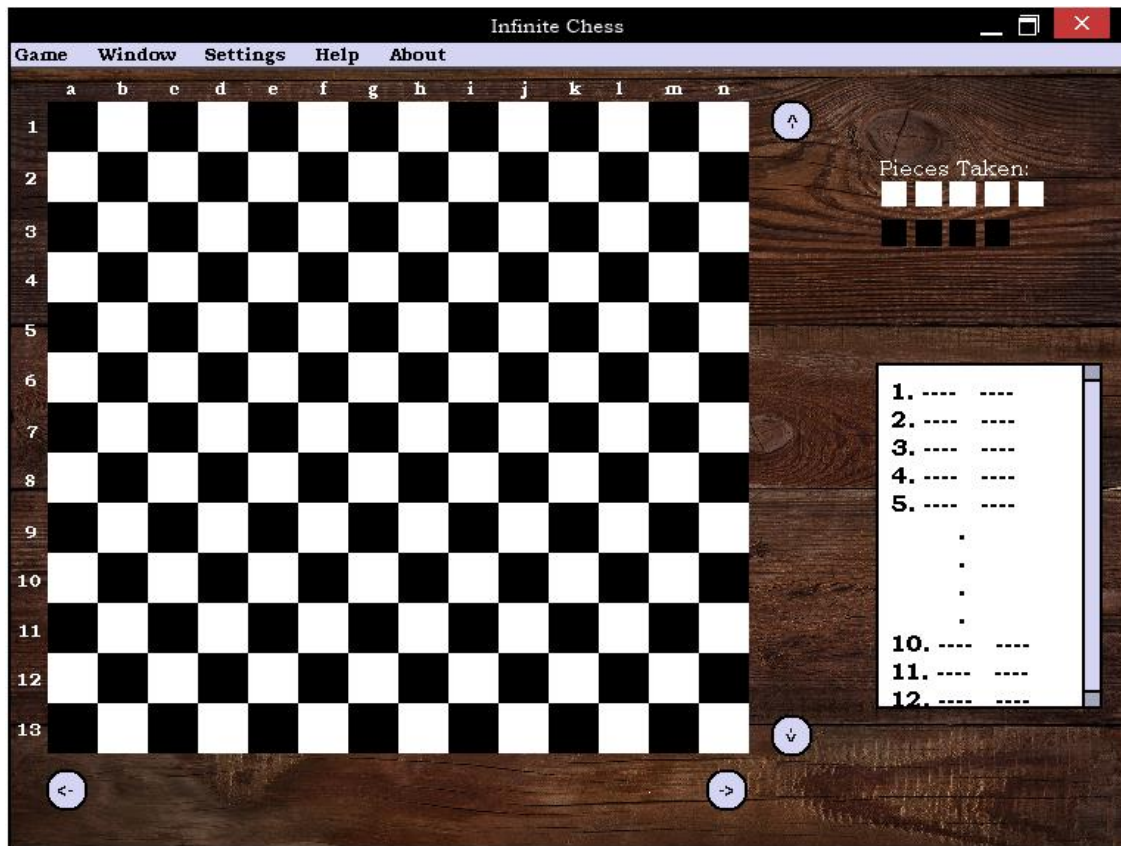
Rather than assigning static values to each piece, we could add some variance based on the environment of the piece. In regular chess, this can be done by defining an 8x8 table of values, and then multiplying the value of the piece by the corresponding value in the table when the piece is on that square. In infinite chess, it is not possible to define an infinite table, so any dynamic values will need to be based on the pieces



nearby. For example, a queen protected by a rook could be worth 10 instead of 9, which would cause the AI to play in such a way that its queens won't be easily taken.

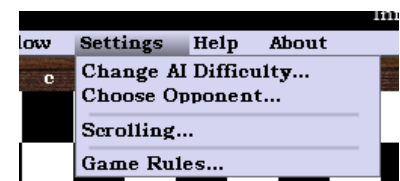
## Window Design

The features that will be needed on the window are the board, scroll buttons, toolbar, move history, pieces taken and labels. Below is a design for the window.



In this design, the visible board is 13 by 14 squares. This is subject to change, as I will need to test what size squares will give the best balance between graphical quality and amount of information available.

The move history is a scrolling textbox seen on the bottom right, with 1 move per line. Again, testing will need to be done to adjust the font size so that it is readable, but shows enough information to be useful. Pieces taken will simply be a label with the icons of pieces that have been taken on each side. The scroll buttons are seen to the left and below the board. I feel the positioning and text of the buttons makes what each will do intuitive. The design of the toolbar buttons can be seen on the right; it's just a standard toolbar.

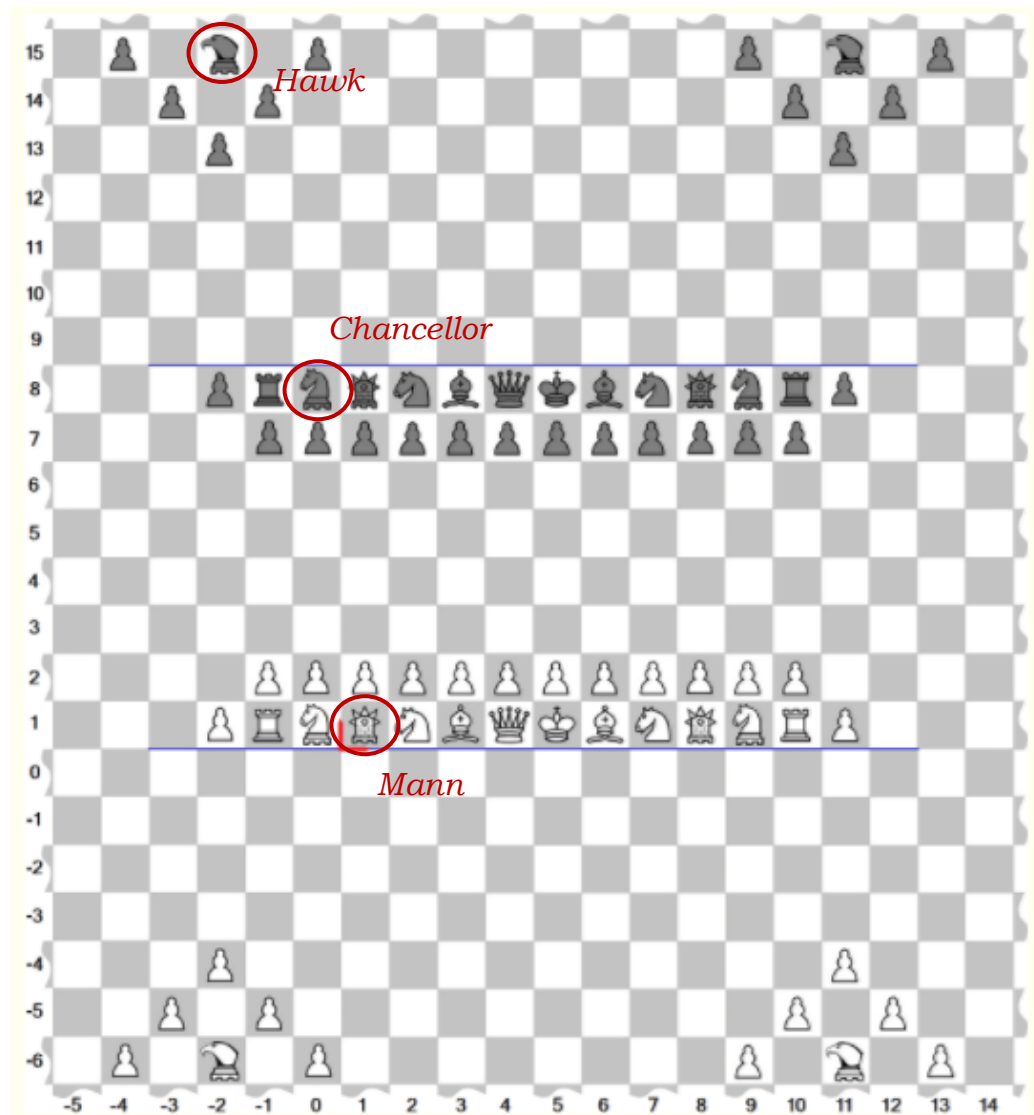


When it comes to changing resolutions, my plan is to have the same ratios for size of each control on the screen (i.e. each control takes up the same amount of space relative to the window size), however, the board will be able to have a higher density of squares. This means that there is an advantage to playing with a higher resolution monitor, rather than everything being static.



## Game Design

The starting configuration for infinite chess is seen below (from chessvariants.com<sup>3</sup>).



This is clearly rather different to the starting configuration for regular chess, and contains some unusual pieces. The piece seen near the corners of the board behind the pawns are known as Hawks, and they move either 2 or 3 squares in any orthogonal direction, and is able to jump over other pieces. One piece inward from the rooks are the Chancellors, which have the movement of a rook and a knight combined. Another piece inwards are the Manns, which have identical movement to a King, but are not affected by check.

Game Rules for Infinite Chess:

- \* Pawn promotion still occurs in this variant of chess. White pawns promote at rank 8, and black at rank 1.
  - Pawns can promote to hawks, chancellors or manns in addition to the regular pieces.
- \* Castling is not a valid action.

- \* The fifty-move rule does not apply.
- \* All other rules are the same as for regular chess.

Move notation differs slightly from regular chess. This is relevant because I will be displaying a move history using the appropriate notation. Examples of notation for infinite chess are as follows:

A queen moves from (6,-3) to (10,-3)	<i>Q(6,-3)-(10,-3)</i>
A rook captures the piece at (5,-2) from (5,6)	<i>R(5,6)x(5,-2)</i>
A pawn promotes to a chancellor at (8,2)	<i>(8,3)-(8,2)C</i>
A knight moving to (5,-4) causes check	<i>N(3,-5)-(5,-4)+</i>

## Testing Overview

As this is an iterative development process, I will be testing code as it is written to ensure it works as expected. A sizeable portion of my code is functional which will allow for easier testing. Visual Studio also has a variable watch feature which will allow me to see the values of my variables while the program is running. Throughout the process, I will screenshot blocks of code as they are written, then test and debug them with input data, expected results and actual result. I will also refactor previous code to become more efficient if necessary or I think of a better way to perform a task.

Example testing:

- \* scrollUp Button
  - Press the button once and observe what happens. The expected result is that all pieces on the board move down by one square, the bounds of the board extend if necessary, and the coordinates of each visible square have changed.
- \* GameContainer.OnMouseClicked()
  - Click on any piece on the board. Output each attribute of this piece to a label on the form. The expected result is the name of the piece type, the square it is on, and the colour of the piece.
- \* Piece.CalculateMovement()
  - Click on a queen piece with various other pieces surrounding it. Observe the squares highlighted. The expected result is there is a line of available moves in each orthogonal direction, up until another piece gets in the way, and no squares highlighted after such obstructions (as seen in the screenshot earlier).

When the game is in a playable state, I can ask my client and other chess players to play against the AI or each other to see if the game plays as it should to them. This black box testing is useful because I am not an experienced chess player, so I am not the best judge of how difficult a given chess AI is to play against.

## Testing Details

For some large features of the program, I will undertake extended testing after development is finished to ensure each section is working. I will also conduct an overall test after development is complete.

### *Scrolling Tests*

After scrolling the board has been implemented, I will test whether it is working correctly. This will involve using a debug label to view the coordinates of a given square. I will then press a certain sequence of scrolling buttons which will be recorded, the expected result will be calculated, and then compared to the observed result. If all the results match, I can be sure that scrolling will work.

For example, the square that is at coordinates 0,0 will be at 1,1 if the scroll up and scroll right buttons are pressed.

### *Initial Piece Movement Tests*

The first iteration of piece movement calculations will not take into account check or checkmate, as these features will be developed after piece movement is completed. This testing will be done as a series of screenshots of different scenarios, with at least one for every piece. For each test, the piece to be clicked will be stated, along with the expected outcome, and then whether the result was expected or not. Screenshots of the actual outcome will also be included.

For example, I would expect the movement of a rook to stop at a piece that blocks its line of movement.

### *Updated Piece Movement Tests*

The final iteration of piece movement tests will take into account check and checkmate. This means that pieces will not be able to move to squares which would put their king in check, and if their king is already in check, only moves which will move it out of check are valid. This will need to be tested to ensure every piece is having movement altered correctly to account for these possibilities.

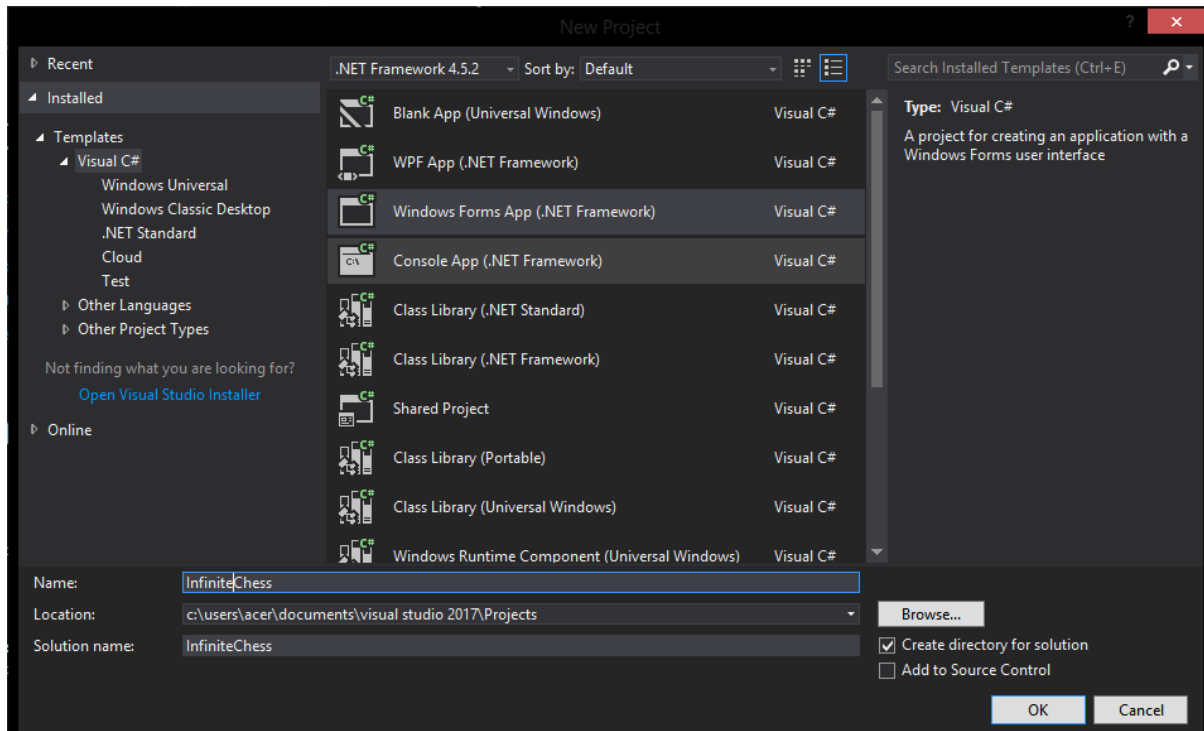
For example, a king with an enemy rook directly above it should not be able to move straight down, since this would not remove it from check.

*\*this section not yet finished\**

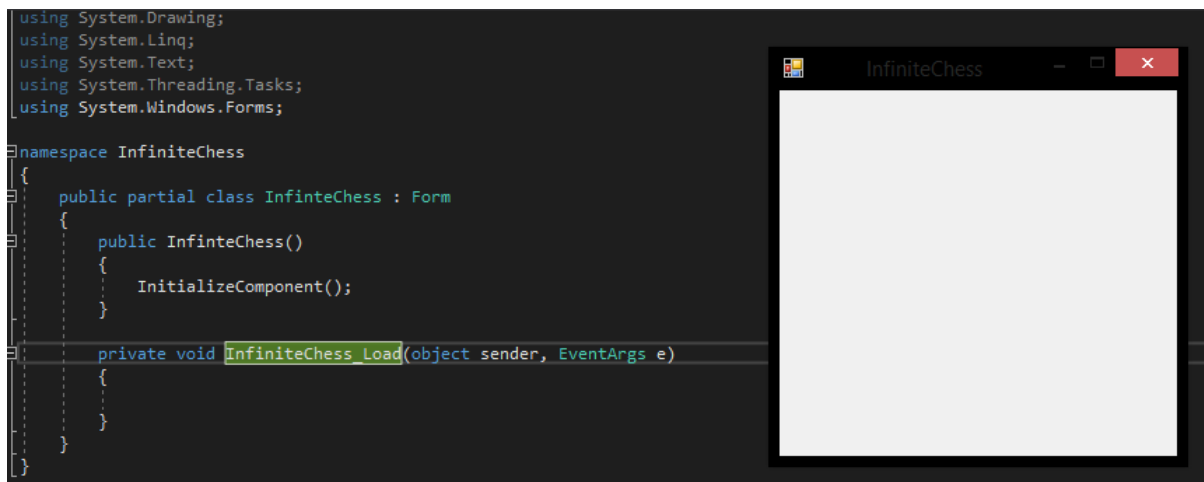
## Section 3: Development

### Project Creation

I am using Visual Studio to develop this game. The first step is to create a new Windows Forms Project.



This creates an `InfiniteChess : Form` class, which is just an empty window with no components. This is what I will build the game upon.



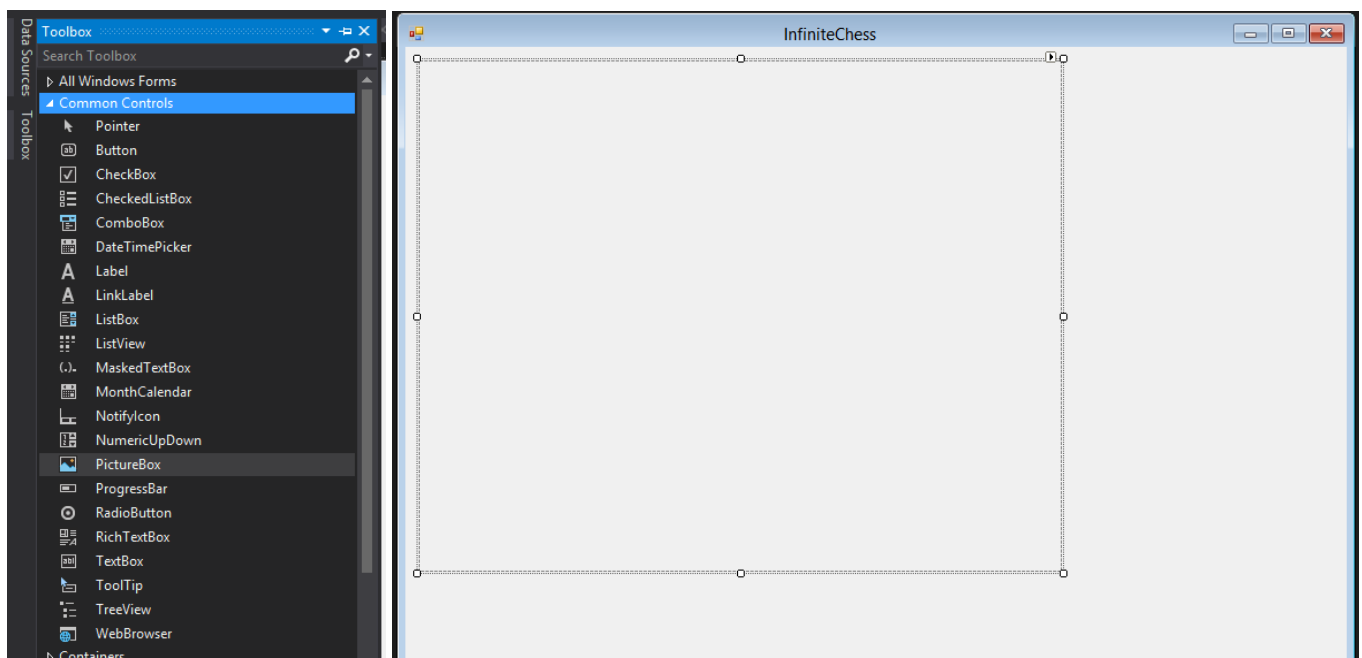
## Creating the Board

The first thing I need to figure out is how to actually get a chess board onto the window. I will change the size of the window to 864x720 so that it is big enough to display an image, but not so big that it cannot fit onto the 720p screen of my laptop.

One approach I could use is draw a board manually using `System.Graphics`, and drawing each square on the board separately. This would need to be stored as an array of `Rectangles` (a built-in type for graphics). However, I also need to store another array of `Squares` anyway to hold the other data for each square of the board. This would mean I have to work with two separate arrays, one holding graphics and the other data, which need to be able to be linked to each other.

However, this is unnecessarily complicated if you consider the movement of the board itself. The only difference between this program and regular chess when it comes to the board is the need for it to scroll. If you were to scroll the board in any direction by one unit, every square you can see would take on the opposite colour. As a result, moving two units in any direction will leave the board visually unchanged. This means that if any scrolling is done in multiples of two units, the board will always look the same, and therefore the board does not need to be redrawn on every scroll. So instead of constructing a board manually, drawing an entire image from a file will increase the efficiency of the program and reduce the resources required.

To get an image onto the window, I will need a `PictureBox` control, which I can select from the Toolbox and drag onto the window.



The window now has an empty `PictureBox` on it. The image displayed is a property of the control; to set this property I can set it in the visual editor to an image, or I can set

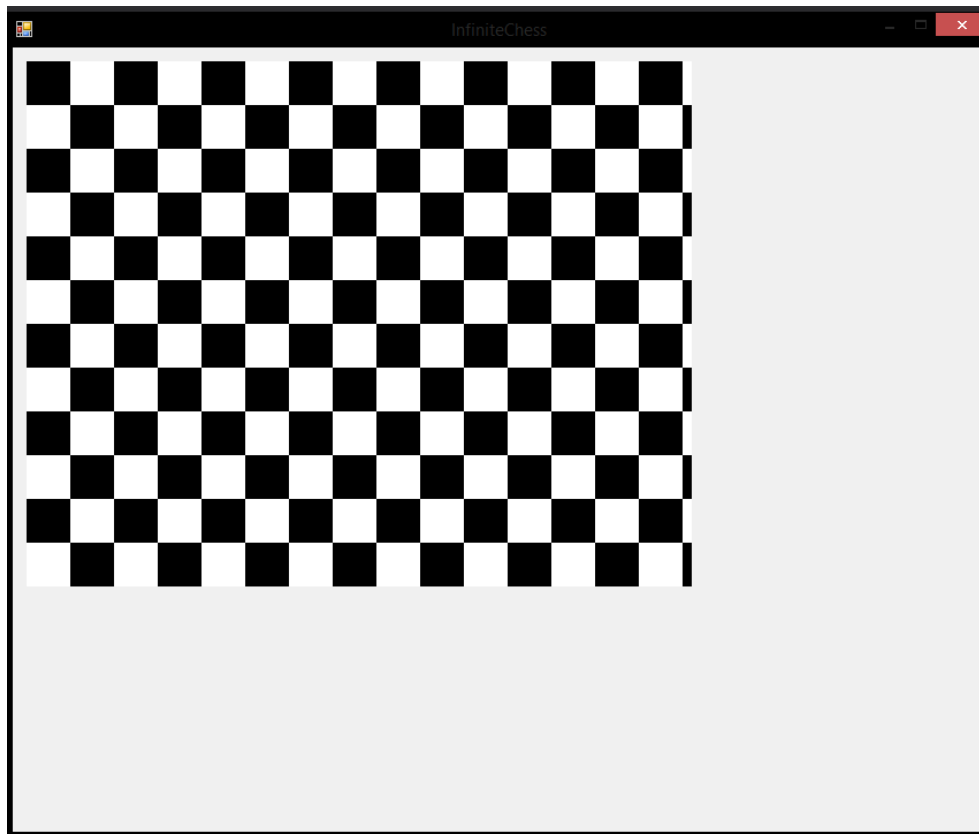
```
private void InfiniteChess_Load(object sender, EventArgs e)
{
    board.Image = new Bitmap("res/image/board.png" + "");
}
```

it within the code. Since I may want to change the image later on during program

execution, it would be more logical to write the code for it. I have named this instance `board`, so I can now use the following code to set its image.

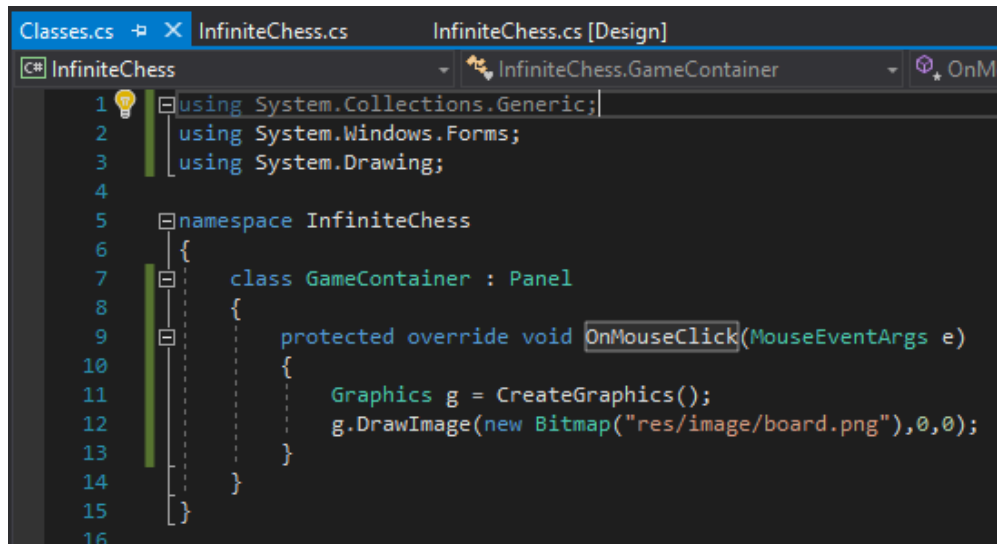
`Bitmap` is a built-in class which handles images. In this case, I am using its constructor which takes a file path as an argument. `res` is a folder I have created to house all the resources the game will need (further on graphics for things such as pieces will be needed). `board.png` is an image file I created which is just a simple checkerboard pattern. `InfiniteChess_Load` is an automatically generated method for the event which is called after the window has finished loading. Windows forms programming is largely event-driven; double-clicking on a control in the design view adds a new method which is called when the main event tied to that control occurs. For example, double clicking on a `Button` will generate a method which is called when the button is clicked.

Running the code as it is now causes the following window to appear. This is not quite what one would expect a chess board to look like; some of the squares have been cut off.



At this point, I have also realised another problem with what I have done so far. If I use a `PictureBox` for the board, I will not be able to draw pieces over the top of it without using another `PictureBox` for each piece because of how Windows Forms decides which controls are on which layers. To get around this, I will use the `Panel` class instead and will draw my own images using `System.Graphics`. As per my design section, I already know I will want to override the functions `OnMouseMove` and `OnMouseClick` from `Panel`, so I will create a custom class which extends `Panel` called

GameContainer. To make it easier for me to navigate my code, I will create this in a new file called `Classes.cs`, where I will also put my other classes later.



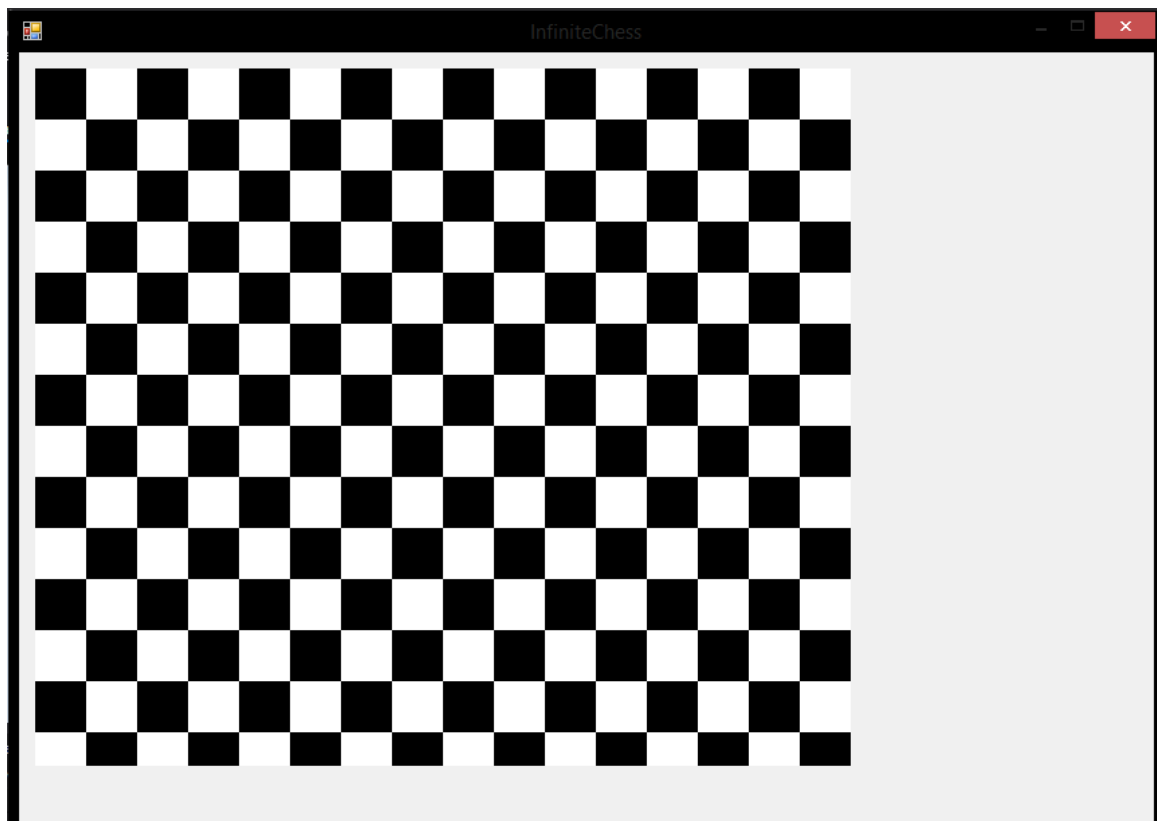
```

Classes.cs  InfiniteChess.cs  InfiniteChess.cs [Design]
C# InfiniteChess  InfiniteChess.GameContainer  OnM
1  using System.Collections.Generic;
2  using System.Windows.Forms;
3  using System.Drawing;
4
5  namespace InfiniteChess
6  {
7      class GameContainer : Panel
8      {
9          protected override void OnMouseClicked(MouseEventArgs e)
10         {
11             Graphics g = CreateGraphics();
12             g.DrawImage(new Bitmap("res/image/board.png"), 0, 0);
13         }
14     }
15 }
16

```

I have overridden the method `OnMouseClicked` to draw the image of the board on the panel at 0,0. This will allow me to test if this class is implemented correctly. I will create a new instance of `GameContainer`, called `boardPanel1`, in the design tab and then run the program. After clicking the panel, an image of the board should appear.

Clicking the panel causes the following to happen:



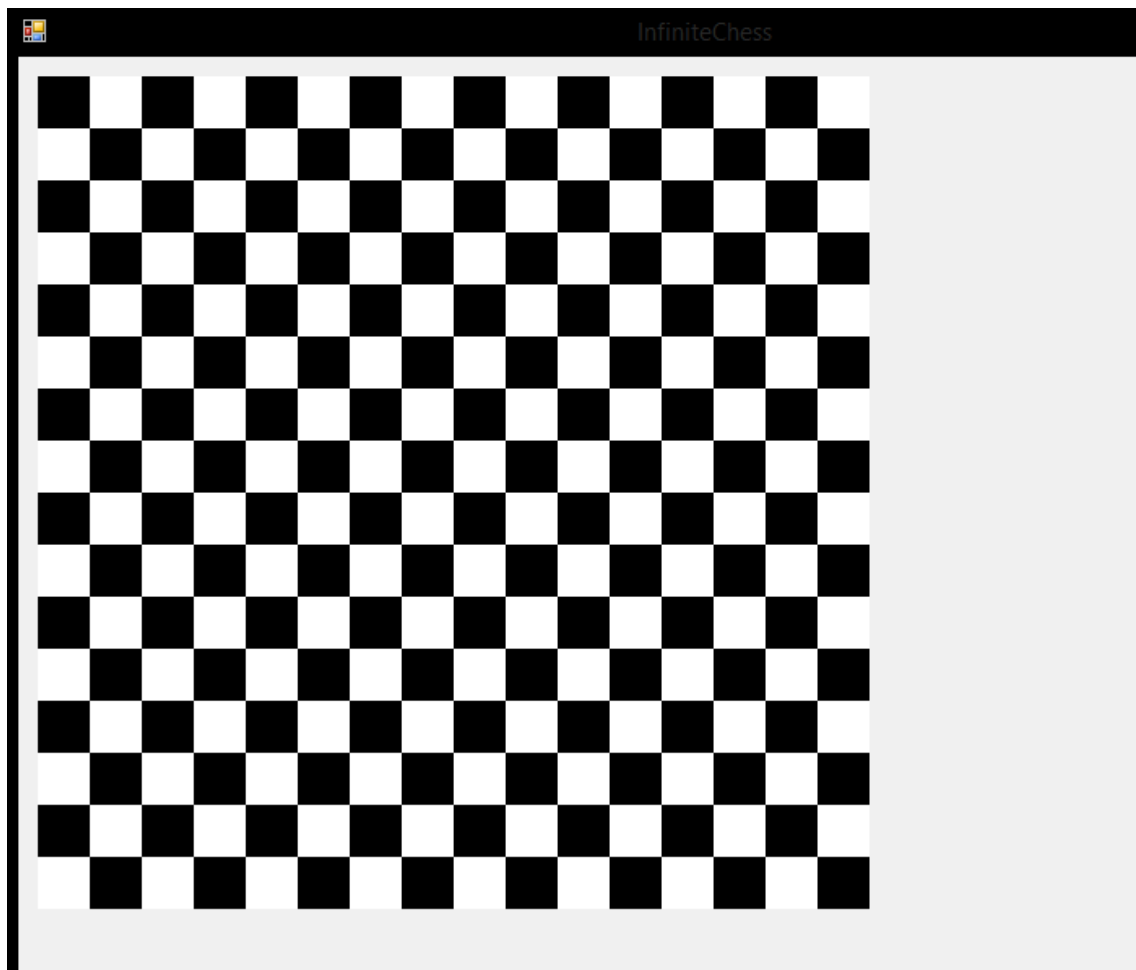
This means the class has been implemented correctly and I can now build upon it.

Currently, the board isn't being drawn quite how I want it; there are still some squares being cut off. However, since I'm using `Graphics` to draw it, I can specify the size of the image to draw. When using `Graphics`, the coordinates used are based off the control the instance of `Graphics` was created in. In this case, the coordinates 0,0 are the top left corner of `boardPanel1` (not the top left corner of the whole window). One convenient property of this is drawing something with coordinates which are outside the control will simply not be drawn. This means that if I specify a size for the image which is larger than the size of `boardPanel1`, I can magnify the board.

For now, I want the board to be 16x16 squares, which is the size of the actual image. If each square is 32x32 pixels, the entire `boardPanel1` will be 512x512, which fits on the current window, so this is the size I will use for now. The previous code will be updated to reflect the new size we want:

```
protected override void OnMouseClicked(MouseEventArgs e)
{
    Graphics g = CreateGraphics();
    g.DrawImage(new Bitmap(new Bitmap("res/image/board.png"), new Size(512,512)), 0, 0);
}
```

Running the program and then clicking the panel yields the following:



This is a 16x16 grid of complete squares with a size of 512x512 pixels.

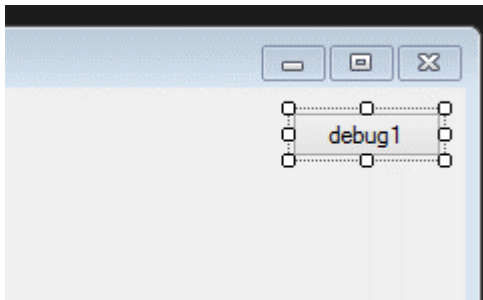


## Adding Board Functionality

Now that I have a grid, a way to store data about the board is needed. As mentioned in the Basic Classes section, I will be using the class `Square` to describe the pixel coordinates of each square. I will create the `Square` class in `Classes.cs`:

```
public class Square
{
    public int X { get; set; }
    public int Y { get; set; } //actual coordinates
    public short indexX { get; set; }
    public short indexY { get; set; } //square reference
    public static List<Square> emptyList() { return new List<Square> { }; }
    public override string ToString()
    {
        return indexX.ToString() + ", " + indexY.ToString() + ", " + X.ToString() + ", " + Y.ToString();
    }
}
```

I now need to create the `InitialiseBoard()` function, which will use a `List<Square>` to represent the board. To do this, two nested `for` loops will be required. Since all this function will do is add some elements to a list, there is no visual feedback that it is working correctly. As a result, I will use `Graphics` to draw something on each square that is created so that I can ensure it is functioning correctly.

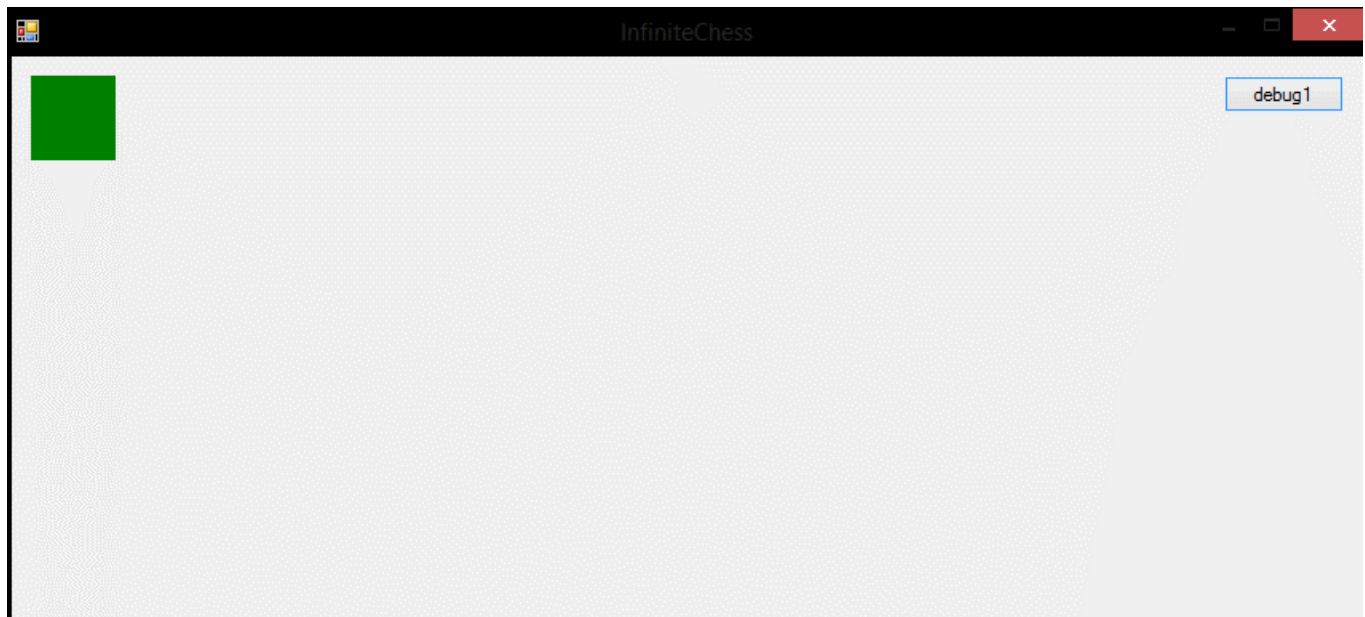


At this point, I will also add a debugging button to the form. This will be useful because I can test out specific functions by just assigning them to this button. For now, I will assign `InitialiseBoard()` to this button, so that whenever I click it, board will be populated.

```
//create the logical board
public void InitialiseBoard() {
    Graphics g = boardPanel.CreateGraphics();
    for (int i = 0; i < 16; i++) { //columns
        for (int j = 0; j < 16; j++) { //rows
            board.Add(new Square { X = 38 * i, Y = 38 * j, indexX = (short)i, indexY = (short)j });
            g.FillRectangle( new SolidBrush(Color.Green), i, j, 38, 38);
        }
    }
    g.Dispose();
}
```

Each loop repeats 16 times, which will create a board of size 16x16 squares. `board.Add()` takes a `Square` as an argument, and the constructor for `Square` takes an x-coordinate, a y-coordinate, an x-index and a y-index. The indexes for each square are simply the `i` and `j` values used in the loops. The coordinates for each `Square` is calculated by multiplying the `i` or `j` value by 38. This should give squares of size 38x38 pixels.

Running the program and pressing the button yields the following:

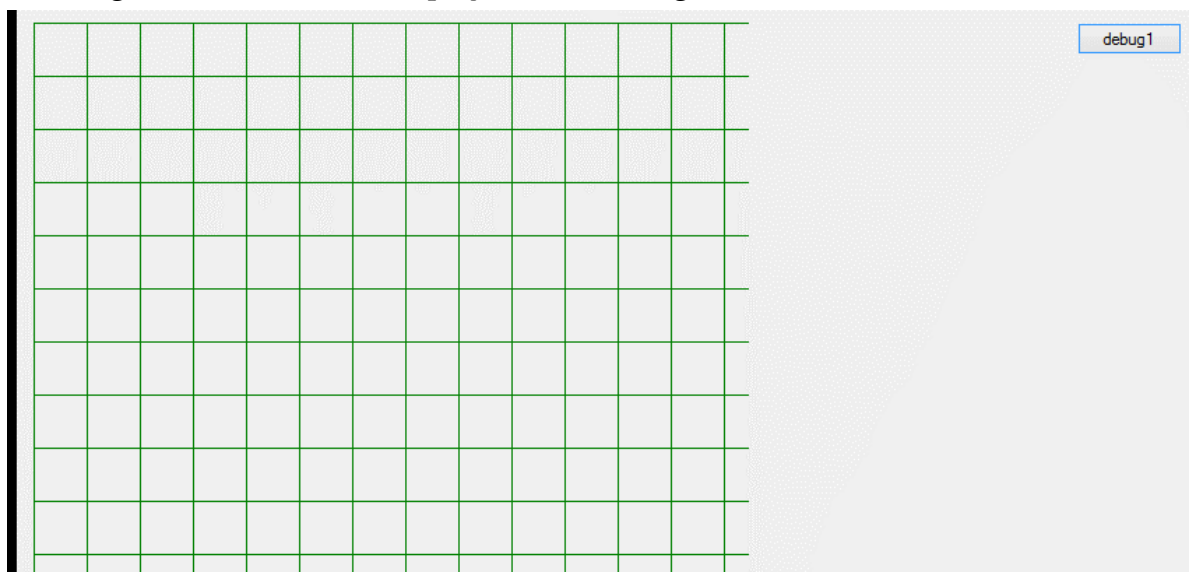


This is not quite the expected outcome. It appears to have only draw a single square. However, this square is clearly larger than  $38 \times 38$  pixels. Looking at my code once again, I see that in `FillRectangle`, I have provided `i`, `j` as the first and second arguments. These arguments define the coordinates of the top left of the shape to draw. Since the maximum value of these 2 arguments is 16, it is drawing the correct number of squares, but all in the top left corner. Multiplying these two arguments by 38 (the size of the squares) should resolve this problem.

Another problem I have noticed is that I cannot distinguish between squares with this method of drawing. This function will just draw a large green square of size  $(16 \times 38)$ , which is not helpful at all. To fix this, updating the last line to draw outlines of rectangles instead of full rectangles will be enough:

```
g.DrawRectangle( new Pen(Color.Green), 38*i, 38*j, 38, 38);
```

Running this function now displays the following:



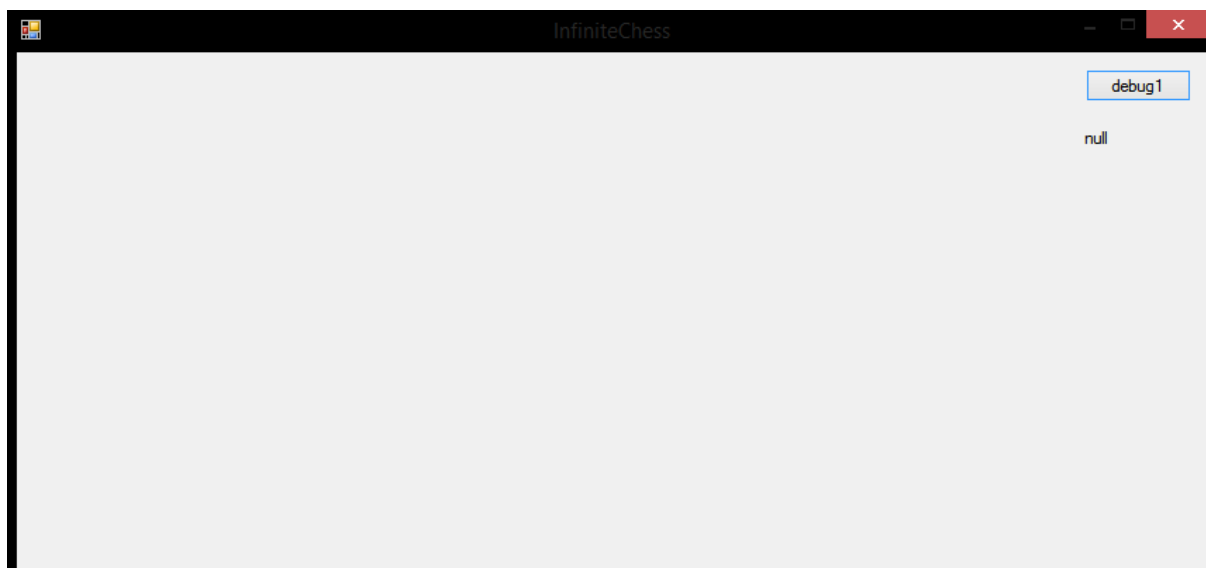
A grid of squares, which is what we wanted. I can see that the structure of the List is correct as a whole, but I still can't tell if each individual square is where I expect it to be. To do this, I will need some way of outputting the attributes of each Square. The best way to do this would be to use a debug label and have the OnMouseMove() function display information about the Square the mouse is over in the label. First, I need a way to find a Square given its coordinates (which are passed in from OnMouseMove()). This will be a function in GameContainer, because it will then be easier to use this function inside other methods in GameContainer.

```
public static Square findSquareByCoords(int x, int y) {
    //iterate through each square
    foreach (Square s in InfinteChess.board) {
        //determine if the passed in coordinates are in the boundaries of the square
        if (x >= s.X && x < s.X + 38 && y >= s.Y && y < s.Y + 38) return s;
    }
    return null;
}
```

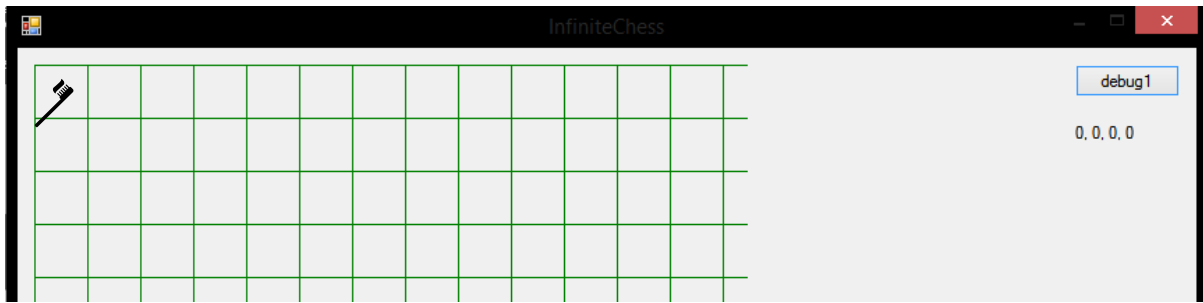
And then to set up the label, which makes use of my overridden Square.ToString():

```
protected override void OnMouseMove(MouseEventArgs e)
{
    Label l = (Label)Parent.Controls.Find("debug2", false)[0];
    Square cursorSquare = findSquareByCoords(e.X, e.Y);
    l.Text = cursorSquare?.ToString() ?? "null";
}
```

This code should display the 4 attributes of the Square the cursor is currently over in the debug label, and it should display "null" if there is no Square where the mouse cursor is. Now I can check if each Square is where I expect it to be.



Before anything has been generated, the label displays null, which is the expected outcome since there are no Squares yet.



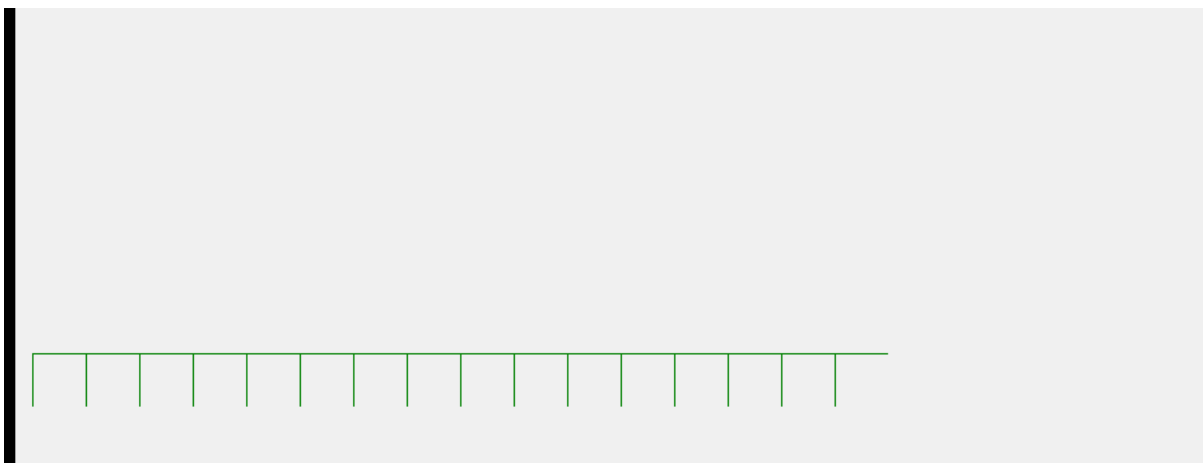
However, there is a problem here. My cursor is over the top left square, but this square has index  $[0,0]$  according to the label. This is because coordinates are labelled from the top left of an object (so this square also has coordinates  $(0,0)$ ), but I want the indexes to start from the bottom left. This means I will need to adjust the code for initialising the board so that  $[0,0]$  is where I want it.

Note: square brackets  $[]$  will be used to refer to an index, parentheses  $()$  will be used to refer to coordinates

The bottom left corner in a  $16 \times 16$  grid is at  $(0,570)$ , so we need to make sure  $[0,0]$  is there. I will store the coordinates  $(0,570)$  as a variable so that I can adjust this later if needed. I can then use the variable in the board generation code to move  $[0,0]$  to  $(0,570)$ .

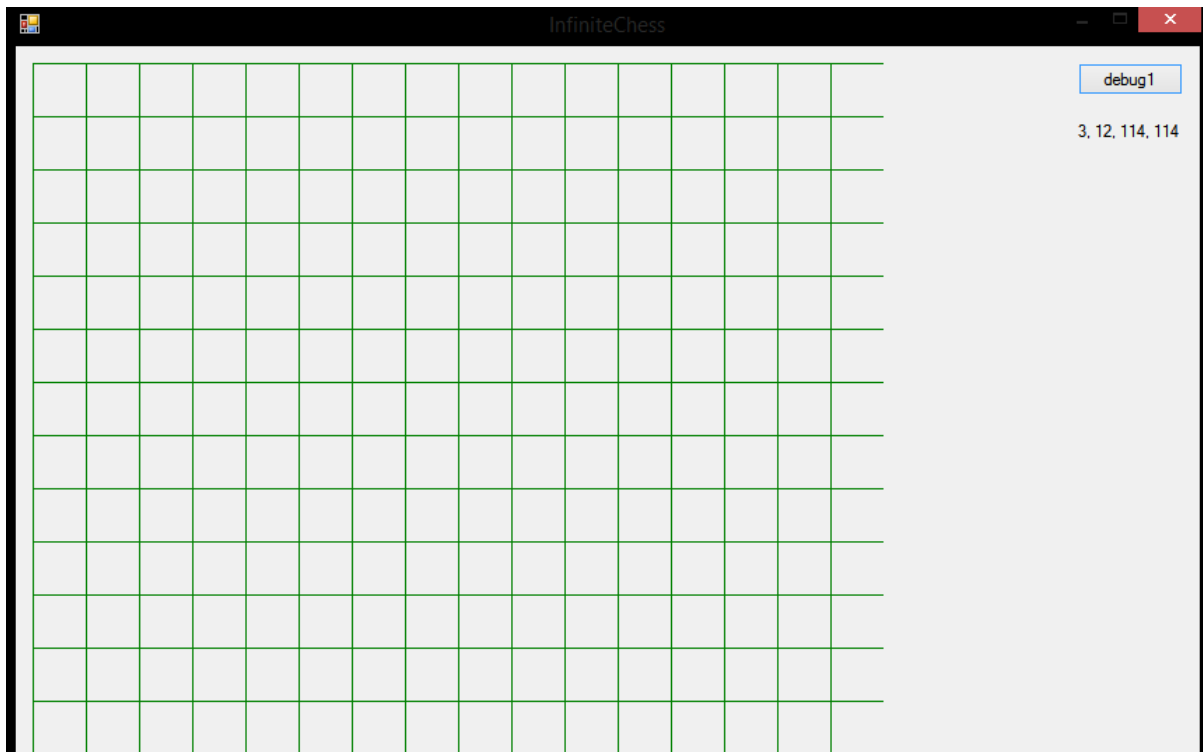
```
board.Add(new Square { X = origin[0] + 38 * i, Y = origin[1] + 38 * j,
```

By adding 570 to each Y coordinate, I effectively move every square down by 16 squares, which means  $[0,0]$  is now at  $(0,570)$ . However, most of the squares are now off the screen since the positive Y direction is still down rather than up, as seen below.



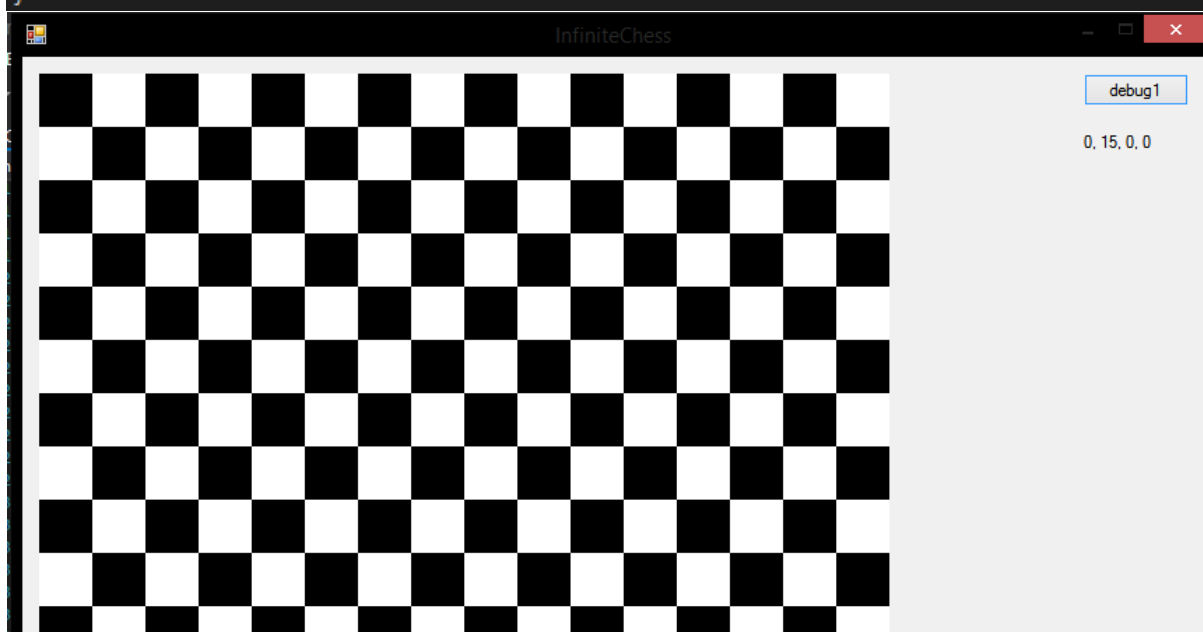
To fix this,  $38*j$  needs to be subtracted from  $origin[1]$  instead of added. This reverses the Y direction for indexes, which will give the desired result;  $[0,0]$  is in the bottom left and  $[15,15]$  is in the top right.

```
board.Add(new Square { X = origin[0] + 38 * i, Y = origin[1] - 38 * j,
```



To complete this, I just have to draw the board image over the Squares, and then I will have a regular chess board of size 16x16:

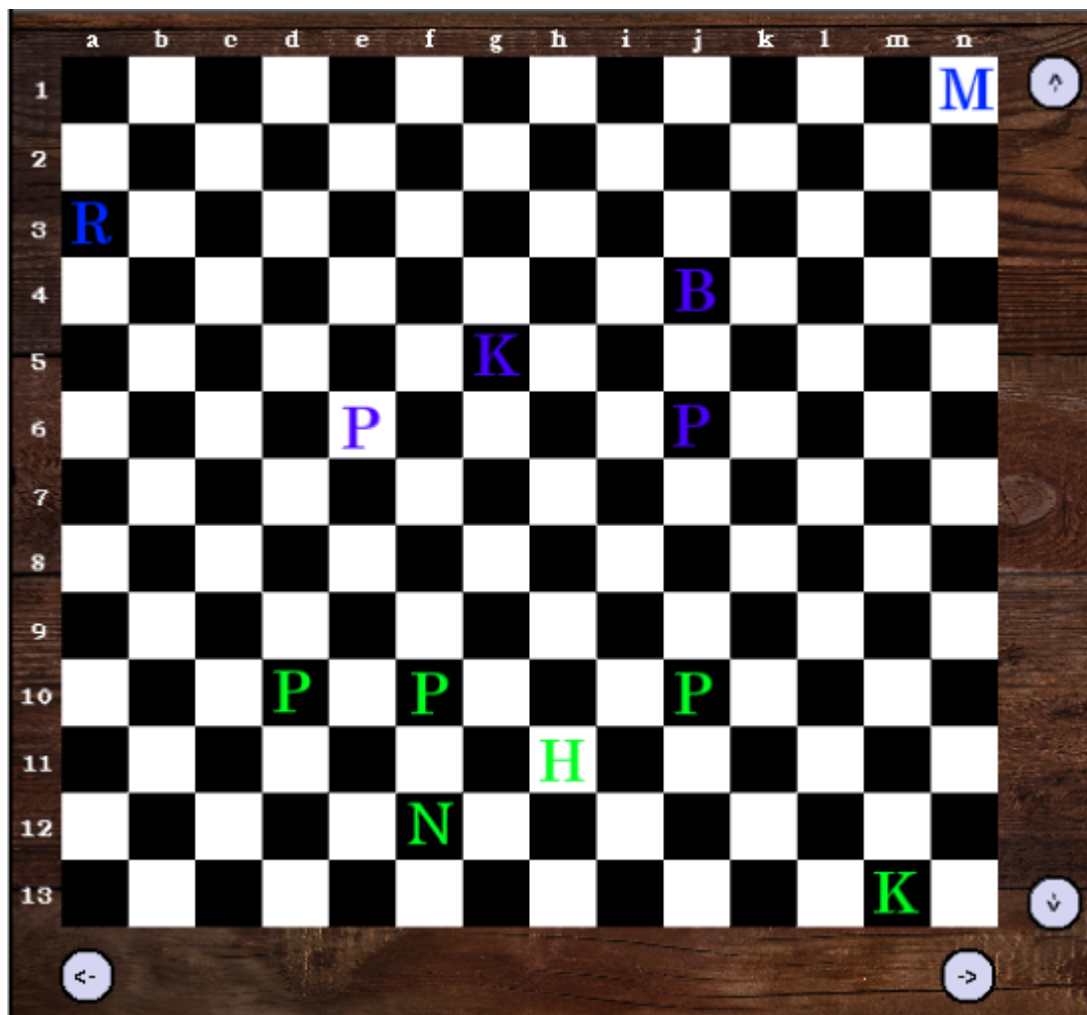
```
public void InitialiseBoard() {
    Graphics g = boardPanel.CreateGraphics();
    g.DrawImage(new Bitmap(new Bitmap("res/image/board.png"), new Size(608, 608)), 0, 0);
    for (int i = 0; i < 16; i++) { //columns
        for (int j = 0; j < 16; j++) { //rows
            board.Add(new Square { X = origin[0] + 38 * i, Y = origin[1] - 38 * j, indexX
            //g.DrawRectangle( new Pen(Color.Green), origin[0] + 38 * i, origin[1] - 38 *
            }
        }
    }
    g.Dispose();
}
```



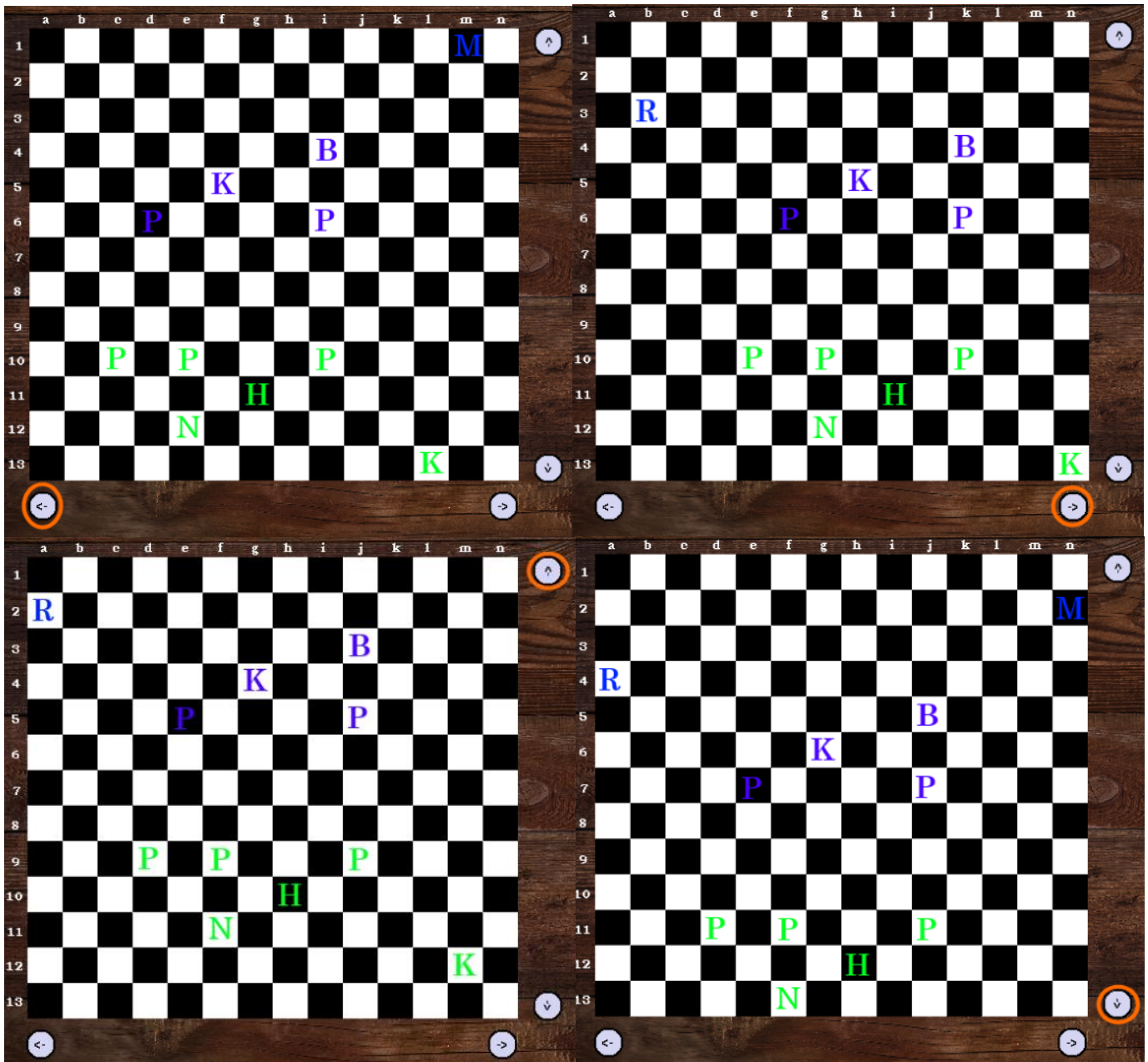
## The Infinite Board

This is infinite chess, not regular chess, so simply drawing a 16x16 grid is not enough to be able to progress further in development. I need to implement a board with an infinite nature, and clearly I cannot display anything with infinite size. This will be circumvented by being able to scroll the board across the screen to access the rest of it. In reality, the board cannot actually be infinite because any computer system only has a finite amount of memory, but it can be made large enough that it will be effectively infinite.

Here is a sample board configuration with the pieces being blue and green, with scroll buttons:



The following images show the new state of the board if the highlighted scroll button is pressed:



As can be seen, scrolling the board can cause pieces to become out of view. They still exist, and scrolling the board back to them will cause them to reappear.

As seen earlier, the board is represented as a `List<Square>`. For the finite 16x16 board, this list contains 256 elements. For the infinite board, it will be much larger, depending on what I decide the size should be. However, defining all these Squares at the start of the program would be an inefficient use of memory, because chances are most of the board will be unused throughout the course of a game. For this reason, I want to begin with the board being 16x16 and add new Squares as the board is scrolled.



As mentioned in Basic Classes, I will use two variables to keep track of the position of the board: `int[] bounds` and `int[] origin`. These will keep track of the position of the edges of the board and the coordinates of [0,0], respectively.

`bounds` is used to solve the earlier problem with efficient use of memory. This is done by using an algorithm similar to the one below:

```
public void BoardScrollUp() {
    //check if the square in the top left position on the visible board has the
    same Y coordinate as the currently stored boundary for the top of the board
    if square at (0,0) has Y coordinate equal to upper Y bound {
        //if it does, that means we are at the edge of the board and need to make
        a new row of squares
        for (int i = left X bound; i < right X bound; i++) {
            add a new square at [current X, upper Y bound + 1]
        }
    }
}
```

`origin` simply stores the coordinates of [0,0], and using this I will be able to calculate which squares should be showing on the screen at any time.

I will now add these two variables to my list of global variables, and while I'm there I will also create a `size` variable to store the size of the visible section of the board, so that this doesn't become hard-coded into the game. Doing this also means I can have the boundaries be initialised to match the size of the visible board: if the initial size is 16x16 then the upper Y boundary would be 15, but if the initial size was 10x10, the upper Y boundary would need to be 9. I will add the code to do this automatically in the constructor for the form itself.

```
//global variables
public static List<Square> board = new List<Square>(); //stores all squares of the board
public static int[] origin = { 0, 570 }; //coordinates of [0,0]
public static int[] bounds = { 0, 0, 0, 0 }; //boundaries of the generated board
public static int[] size = { 16, 16 }; //size of the visible board

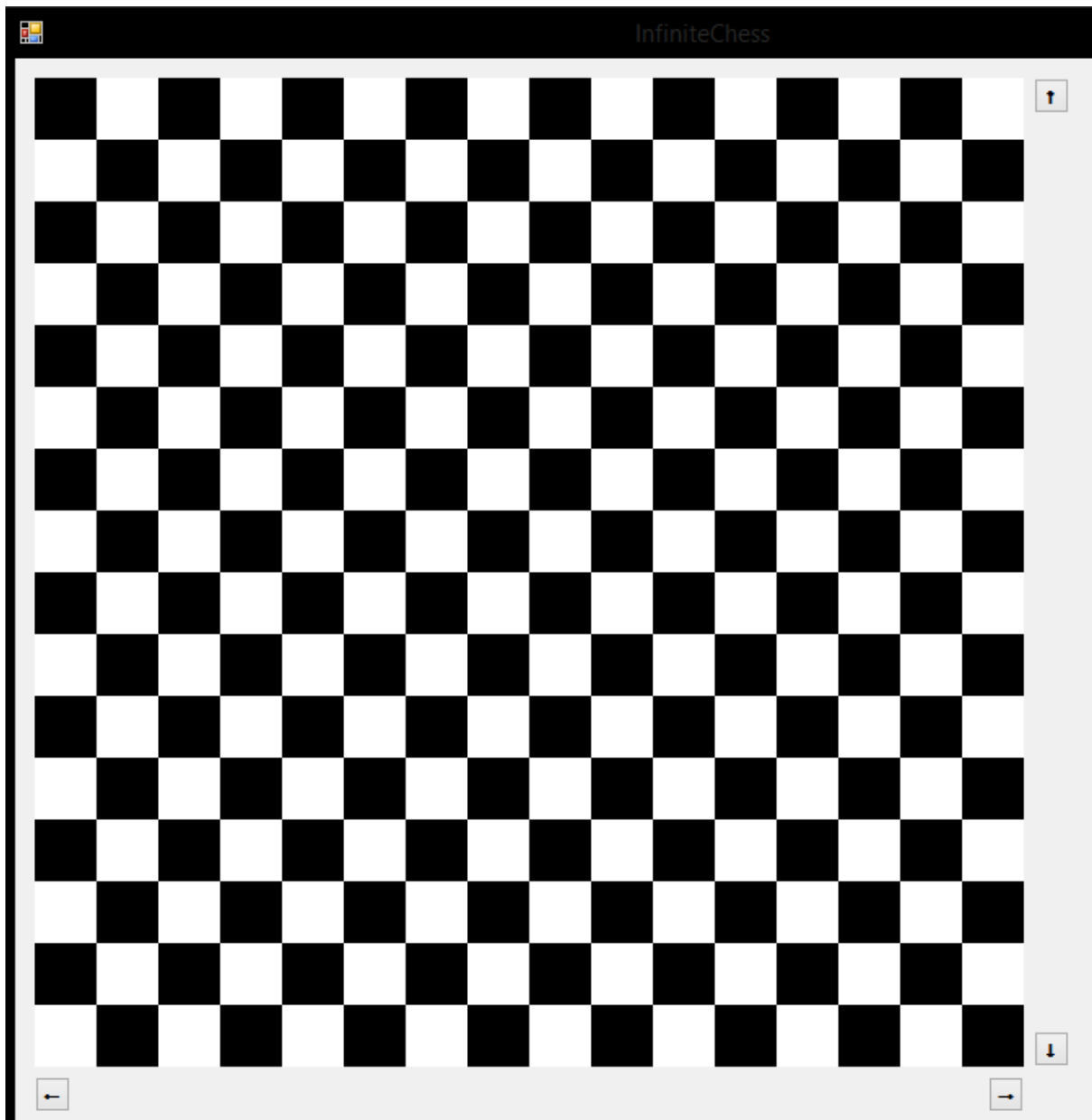
public InfinteChess()
{
    InitializeComponent();
    bounds[1] = (size[0] - 1); bounds[3] = (size[1] - 1); //initialises the boundaries
}
```

Now that I have a way to keep track of a larger board, I can add scrolling functionality. While scroll bars are an option in Windows Forms, because the board has a dynamic (and large) size, these will not be a suitable option. The position of the bar would change each time the board increases in size, and when the board becomes very large even the smallest scroll would move many squares, making it difficult to scroll the board precisely where the user wants.

Instead, I will use buttons that can be clicked to scroll in a certain direction. These buttons will be called `sUp`, `sDown`, `sRight` and `sLeft`, and each will feature a text arrow



indicating the direction the button will scroll the board. After adding them to the window, the game looks like this:



The code for each button will be largely similar, so I will test the system with the scroll up button (sup) first, and then copy and adapt the code for each of the other buttons. For this, I will use another debug label which will output the current value of bounds.

Using the algorithm from above, I have written the following code:

```
private void sUp_Click(object sender, EventArgs e)
{
    Square edge = GameContainer.findSquareByCoords(0, 0);
    if (edge.indexY == bounds[3]) {
        bounds[3]++;
        for (int j = bounds[0]; j <= bounds[1]; j++) {
            board.Add(new Square { X = origin[0] + 38 * j,
                                   Y = origin[1] - bounds[3] * 38,
                                   indexX = (short)j,
                                   indexY = (short)bounds[3] });
        }
    }
    debug3.Text = $"{bounds[0]},{bounds[1]},{bounds[2]},{bounds[3]}";
}
```

After running the program, the label initially displays 0,15,0,15, which is the expected value. After pressing the scroll up button, this changes to 0,15,0,16, which is also what I'd expect, since the board starts as 16x16 and I am trying to scroll up. However, this function is not quite complete; while pressing scroll will update boundaries and generate new squares if necessary, the squares that are currently visible do not change. For example, before pressing this button, the square in the bottom left corner was [0,0]. After scrolling up, I would expect this to become [0,1], which does not happen (it stays as [0,0]). This is because I need to update each Square in board so that they have the new correct coordinates. I will create a new function, `updateSquares` to update the coordinates of each Square when the board is scrolled, and then call this in `sUp_Click`:

```
private void sUp_Click(object sender, EventArgs e)
{
    Square edge = GameContainer.findSquareByCoords(0, 0);
    updateSquares(1, false, sender);
    if (edge.indexY == bounds[3]) {
        bounds[3]++;
        for (int j = bounds[0]; j <= bounds[1]; j++) {
            board.Add(new Square { X = origin[0] + 38 * j,
                                   Y = origin[1] - bounds[3] * 38,
                                   indexX = (short)j,
                                   indexY = (short)bounds[3] });
        }
    }
    debug3.Text = $"{bounds[0]},{bounds[1]},{bounds[2]},{bounds[3]}";
}

public void updateSquares(int amount, bool isX, object sender)
{
    origin[isX ? 0 : 1] += 38 * amount;
    if (isX) { foreach (Square s in board) { s.X += 38 * amount; } }
    else { foreach (Square s in board) { s.Y += 38 * amount; } }
}
```

This modified code gives the expected result; here is table which shows the values of each debug label when sup is pressed a given number of times:

NUMBER OF PRESSES	TOP-RIGHT SQUARE	BOUNDS
0	[15,15]	0,15,0,15
1	[15,16]	0,15,0,16
2	[15,17]	0,15,0,17
3	[15,18]	0,15,0,18

Given that this code works, I can now copy and adapt it for the other 3 buttons to give scrolling functionality in each direction.

```
private void sDown_Click(object sender, EventArgs e)
{
    Square edge = GameContainer.findSquareByCoords((size[0] - 1) * 38 + 1, (size[1] - 1) * 38 + 1);
    updateSquares(-1, false, sender);
    if (edge.indexY == bounds[2]) {
        bounds[2]--;
        for (int j = bounds[0]; j <= bounds[1]; j++) {
            board.Add(new Square { X = origin[0] + 38 * j,
                                   Y = origin[1] - bounds[2] * 38,
                                   indexX = (short)j,
                                   indexY = (short)bounds[2] });
        }
    }
    debug3.Text = $"{bounds[0]},{bounds[1]},{bounds[2]},{bounds[3]}";
}

private void sRight_Click(object sender, EventArgs e)
{
    Square edge = GameContainer.findSquareByCoords((size[0] - 1) * 38 + 1, (size[1] - 1) * 38 + 1);
    updateSquares(-1, true, sender);
    if (edge.indexX == bounds[1]) {
        bounds[1]++;
        for (int j = bounds[2]; j <= bounds[3]; j++) {
            board.Add(new Square { X = origin[0] + bounds[1] * 38,
                                   Y = origin[1] - 38 * j,
                                   indexX = (short)bounds[1],
                                   indexY = (short)j });
        }
    }
    debug3.Text = $"{bounds[0]},{bounds[1]},{bounds[2]},{bounds[3]}";
}

private void sLeft_Click(object sender, EventArgs e)
{
    Square edge = GameContainer.findSquareByCoords(0, 0);
    updateSquares(1, true, sender);
    if (edge.indexX == bounds[0]) {
        bounds[0]--;
        for (int j = bounds[2]; j <= bounds[3]; j++) {
            board.Add(new Square { X = origin[0] + bounds[0] * 38,
                                   Y = origin[1] - 38 * j,
                                   indexX = (short)bounds[0],
                                   indexY = (short)j });
        }
    }
    debug3.Text = $"{bounds[0]},{bounds[1]},{bounds[2]},{bounds[3]}";
}
```

The main difference between these functions are the elements of bounds that are being used and modified. Other than that, they are all fairly similar. Since scrolling is now implemented in all directions, I can comprehensively test the scrolling functionality. To do this, I will define a sequence of buttons to press, the expected outcome of this, and then the actual outcome:

- \* **SEQUENCE** refers to the sequence of buttons that was pressed.  $xA$  represents  $x$  scrolls in the direction  $A$ , where  $x$  is a positive integer and  $A$  is a directional arrow. For example,  $2\uparrow$  means scroll upwards twice, and  $3\rightarrow, 1\downarrow$  means scroll to the right 3 times and downwards once.
- \* **EX. ORIGIN** refers to the expected index of the Square at (o,o) after the sequence has been executed. This will be in the form  $[x,y]$ , where  $x$  and  $y$  are integers representing the index of this Square.
- \* **EX. BOUNDS** refers to the expected value of bounds after the sequence has been executed. This will be in the form  $w,x,y,z$ , where  $w$ ,  $x$ ,  $y$  and  $z$  are integers representing the lower x, upper x, lower y and upper y boundaries of the board, respectively.
- \* **ACT. ORIGIN** refers to the actual index of the Square at (o,o). This has the same form as ORGN\_E and will be a screenshot of the debug label.
- \* **ACT. BOUNDS** refers to the actual value of bounds. This has the same form as BNDS\_E and will be a screenshot of the debug label.
- \* The initial index of the Square at (o,o) is  $[0,15]$ . The initial value of bounds is  $0,15,0,15$ .

SEQUENCE	EX. ORIGIN	EX. BOUNDS	ACT. ORIGIN	ACT. BOUNDS
$1\uparrow$	$[0,16]$	$0,15,0,16$	$0, 16,$	$0,15,0,16$
$1\downarrow$	$[0,14]$	$0,15,-1,15$	$0, 14,$	$0,15,-1,15$
$1\rightarrow$	$[1,15]$	$0,16,0,15$	$1, 15,$	$0,16,0,15$
$1\leftarrow$	$[-1,15]$	$-1,15,0,15$	$-1, 15,$	$-1,15,0,15$
$2\leftarrow$	$[-2,15]$	$-2,15,0,15$	$-2, 15,$	$-2,15,0,15$
$5\downarrow$	$[0,10]$	$0,15,-5,15$	$0, 10,$	$0,15,-5,15$
$1\uparrow, 1\rightarrow$	$[1,16]$	$0,16,0,16$	$1, 16,$	$0,16,0,16$
$1\uparrow, 1\leftarrow, 2\rightarrow, 2\downarrow$	$[1,14]$	$-1,16,-1,16$	$1, 14,$	$-1,16,-1,16$
$3\downarrow, 1\rightarrow, 5\uparrow$	$[1,17]$	$0,16,-3,17$	$1, 17,$	$0,16,-3,17$
$4\rightarrow, 4\downarrow$	$[4,11]$	$0,19,-4,15$	$4, 11,$	$0,19,-4,15$
$8\leftarrow, 1\downarrow$	$[-8,14]$	$-8,15,-1,15$	$-8, 14,$	$-8,15,-1,15$
$4\rightarrow, 6\uparrow, 8\leftarrow$	$[-4,21]$	$-4,19,0,21$	$-4, 21,$	$-4,19,0,21$

All the results match up with the expected values, which means board scrolling works and is finished.

## The Piece Class

The next thing to implement is the `Piece` class, which represents the framework for pieces in the game and their properties. This class is outlined in the basic class section already, so implementing a basic framework is simple enough. I will create this in a new file, `Pieces.cs`, to keep it better organised.

```
namespace InfiniteChess
{
    public enum PieceType { PAWN, KNIGHT, ROOK, BISHOP, QUEEN, KING, MANN, HAWK, CHANCELLOR, NONE };
    public enum PieceColour { BLACK, WHITE }; //WHITE moves up the board, BLACK down

    public class Piece
    {
        public PieceType type { get; private set; }
        public Square square { get; private set; }
        public Bitmap icon { get; private set; }
        public PieceColour colour { get; private set; }

        public Piece(PieceType t, Square s, PieceColour c) {

        }
    }
}
```

The constructor for `Piece` will require initialisation of `icon`, which is the graphic that piece will use. I will need to create or find some graphics for each piece so that I can start using pieces.

For now, I have made some basic images which are just a letter to represent each piece. To the right can be seen a black hawk, black queen, white pawn and white bishop. I created a new folder called **HQPB** images which is itself in `res`. This then two folders, `black` and `white`, which contain the graphics for each piece in that colour.

I can now fill in the constructor so that I can start using pieces:

```
public Piece(PieceType t, Square s, PieceColour c) {
    type = t; colour = c; square = s;
    icon = new Bitmap($"res/image/{c.ToString()}/{t.ToString()}.png");
}
```

I will now add a new attribute to `InfiniteChess`, which will be `List<Piece> pieces`, as mentioned in basic classes. I will use this list to hold all the pieces that are currently in the game. I am using a list rather than an array because the number of pieces could change throughout the course of the game (arrays need a defined size on initialisation).

```
public static List<Piece> pieces = new List<Piece>();
```

To test this class, I will need to draw some pieces to the board. To do this, I will first create a method which will initialise a list of pieces with 1 of each type of piece in each colour. This method will be in the `Piece` class itself.

```

public static List<Piece> IntializePieces()
{
    List<Piece> pieces = new List<Piece>();
    for (int i = 0; i < 2; i++) {
        var w = i == 0 ? PieceColour.WHITE : PieceColour.BLACK;
        pieces.AddRange(new List<Piece> {
            new Piece(PieceType.PAWN, GameContainer.findSquareByIndex(3, 4 + i), w),
            new Piece(PieceType.KNIGHT, GameContainer.findSquareByIndex(4, 4 + i), w),
            new Piece(PieceType.ROOK, GameContainer.findSquareByIndex(5, 4 + i), w),
            new Piece(PieceType.BISHOP, GameContainer.findSquareByIndex(6, 4 + i), w),
            new Piece(PieceType.QUEEN, GameContainer.findSquareByIndex(7, 4 + i), w),
            new Piece(PieceType.KING, GameContainer.findSquareByIndex(8, 4 + i), w),
            new Piece(PieceType.HAWK, GameContainer.findSquareByIndex(9, 4 + i), w),
            new Piece(PieceType.CHANCELLOR, GameContainer.findSquareByIndex(10, 4 + i), w),
            new Piece(PieceType.MANN, GameContainer.findSquareByIndex(11, 4 + i), w),
            new Piece(PieceType.NONE, GameContainer.findSquareByIndex(12, 4 + i), w) });
    }
    return pieces;
}

```

I can call this method when the game starts and set the value of pieces equal to the value this function returns. Later, I could move this list a configuration file rather than being hard coded into the program, but for testing purposes this will suffice. The final thing that has to be done to use these is to actually draw them.

The current function to draw the board is `InitialiseBoard()`, which both draws the board background image and creates the list of squares which represents the board. There is no concept of layers in the graphics here, so if I want to move something on-screen, I have to redraw everything. This means it is necessary to call a function that redraws the board and pieces every time something needs to move. `InitialiseBoard()` is not currently suitable for this because it also initialises the list of squares, which is not something I want to happen each time a piece moves. For this reason, I will split up this function into an initialisation section and a drawing section:

```

//create the logical board
public void InitialiseBoard() {
    for (int i = 0; i < 16; i++) { //columns
        for (int j = 0; j < 16; j++) { //rows
            board.Add(new Square {
                X = origin[0] + 38 * i,
                Y = origin[1] - 38 * j,
                indexX = (short)i,
                indexY = (short)j });
            //g.DrawRectangle( new Pen(Color.Green), origin[0] + 38 * i, origin[1] - 38 * j, 38, 38);
        }
    }
    drawBoard();
}

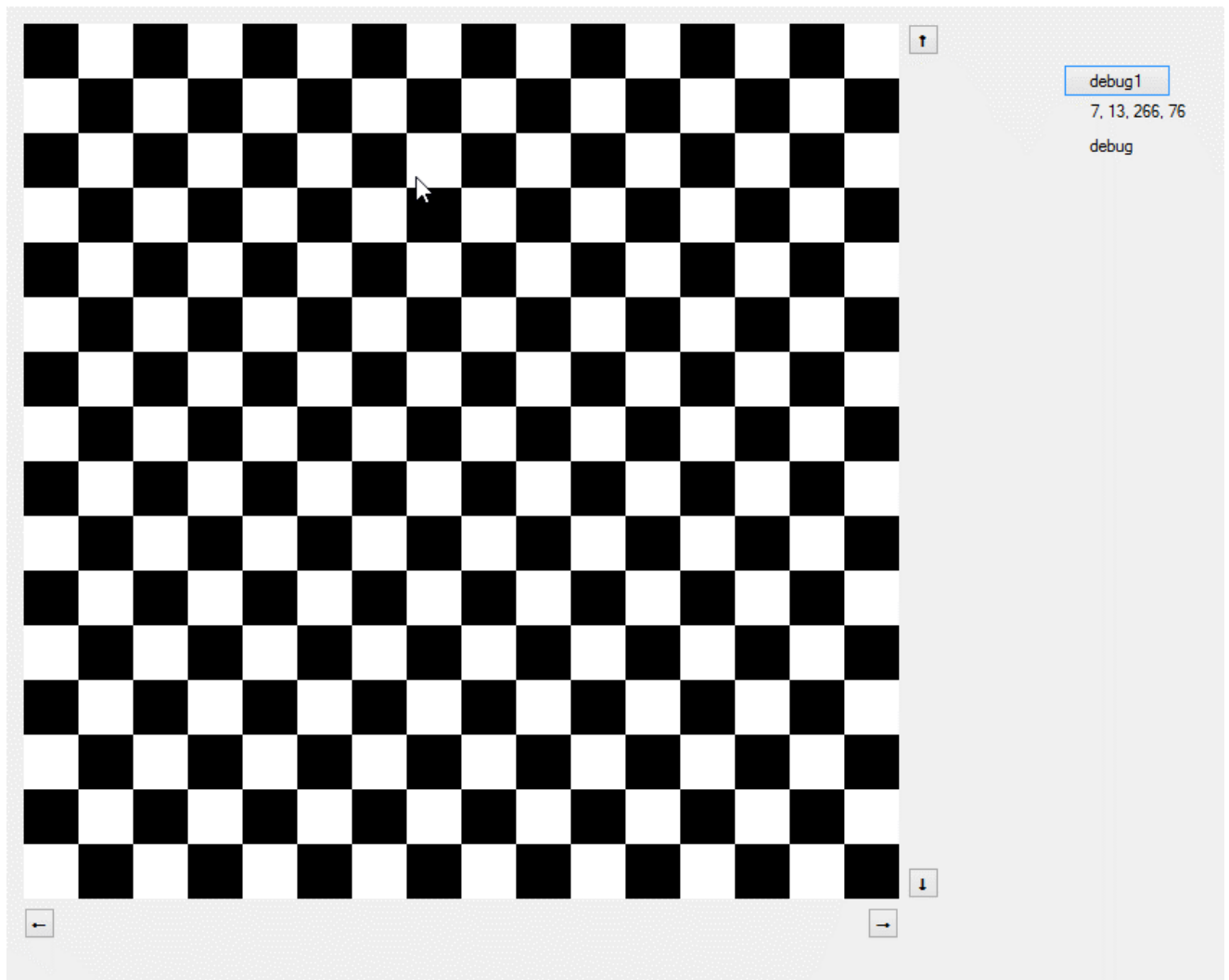
public void drawBoard() {
    Graphics g = boardPanel.CreateGraphics();
    g.DrawImage(new Bitmap(new Bitmap("res/image/board.png"), new Size(608, 608)), 0, 0);
    g.Dispose();
}

```

Now I have a clear distinction between the function to call for initialisation and the function to call for redrawing. All I need to do now is insert some code to draw each piece in pieces into drawBoard().

```
public void drawBoard() {
    Graphics g = boardPanel.CreateGraphics();
    g.DrawImage(new Bitmap(new Bitmap("res/image/board.png"), new Size(608, 608)), 0, 0);
    foreach (Piece p in pieces) {
        Bitmap b = new Bitmap(p.icon, new Size(38, 38));
        g.DrawImage(b, p.square.X, p.square.Y);
    }
    g.Dispose();
}
```

Running the program:



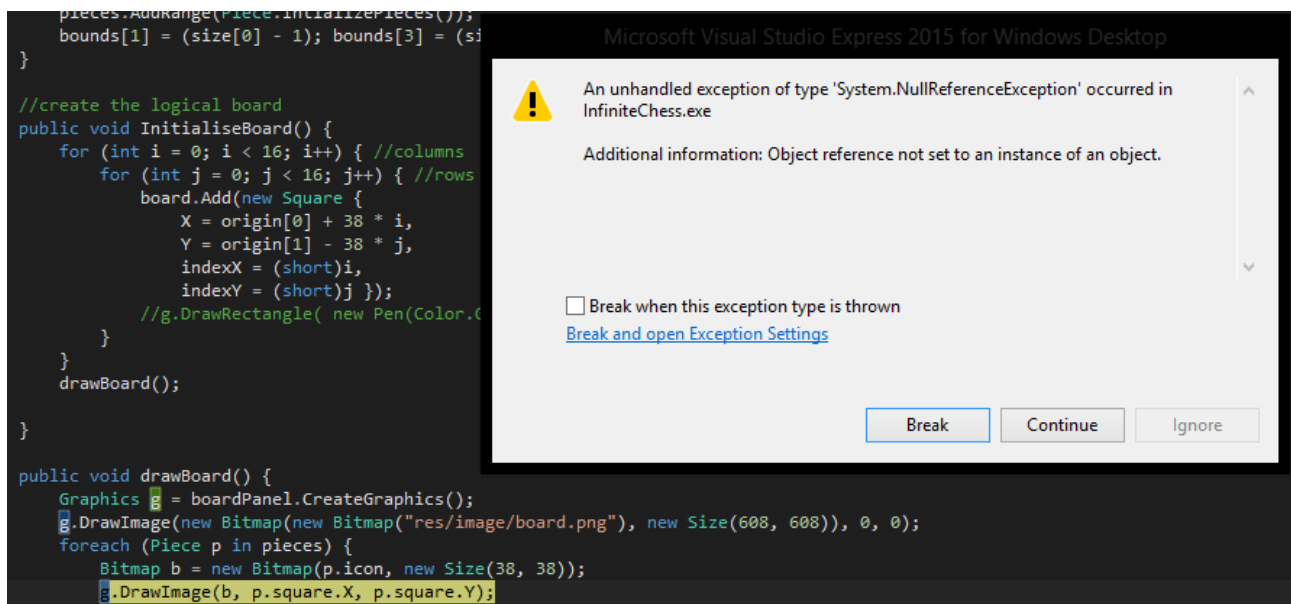
Nothing happens. The reason for this is that even though I created a function which will initialise a `List<Piece>` with some testing pieces, I did not set the actual pieces variable to be equal to this. I add this to the initialise function and try again:

```

public InfiniteChess()
{
    InitializeComponent();
    pieces.AddRange(Piece.InitializePieces());
    bounds[1] = (size[0] - 1); bounds[3] = (size[1] - 1); //initialises the boundaries
}

```

But this happens:



A null reference exception occurs. The line where the exception occurred is that in which the piece is drawn, which means that one of the arguments passed into the function `DrawImage` must have been null. Visual Studio has a helpful feature which allows me to see the values of variables at the moment an exception is thrown. The list looks like this:

Name	Value	Type
▶ this	{InfiniteChess.InfiniteChess, Text: InfiniteChess}	InfiniteC
▶ g	{System.Drawing.Graphics}	System.I
▶ p	{InfiniteChess.Piece}	InfiniteC
colour	WHITE	InfiniteC
icon	{System.Drawing.Bitmap}	System.I
square	null	InfiniteC
@type	PAWN	InfiniteC
▶ b	{System.Drawing.Bitmap}	System.I

`p` is the current piece in the loop (`foreach(Piece p in pieces)`) that is going to be drawn. Looking at the attributed of `p`, we can see that `square` has the value `null`. This means that when the `Square` associated with the piece was created, one of its attributes was also `null`. Looking at the piece initialisation function again (see above), the function used to attach pieces to squares is `GameContainer.findSquareByIndex`. If we take a look at the code for this function:

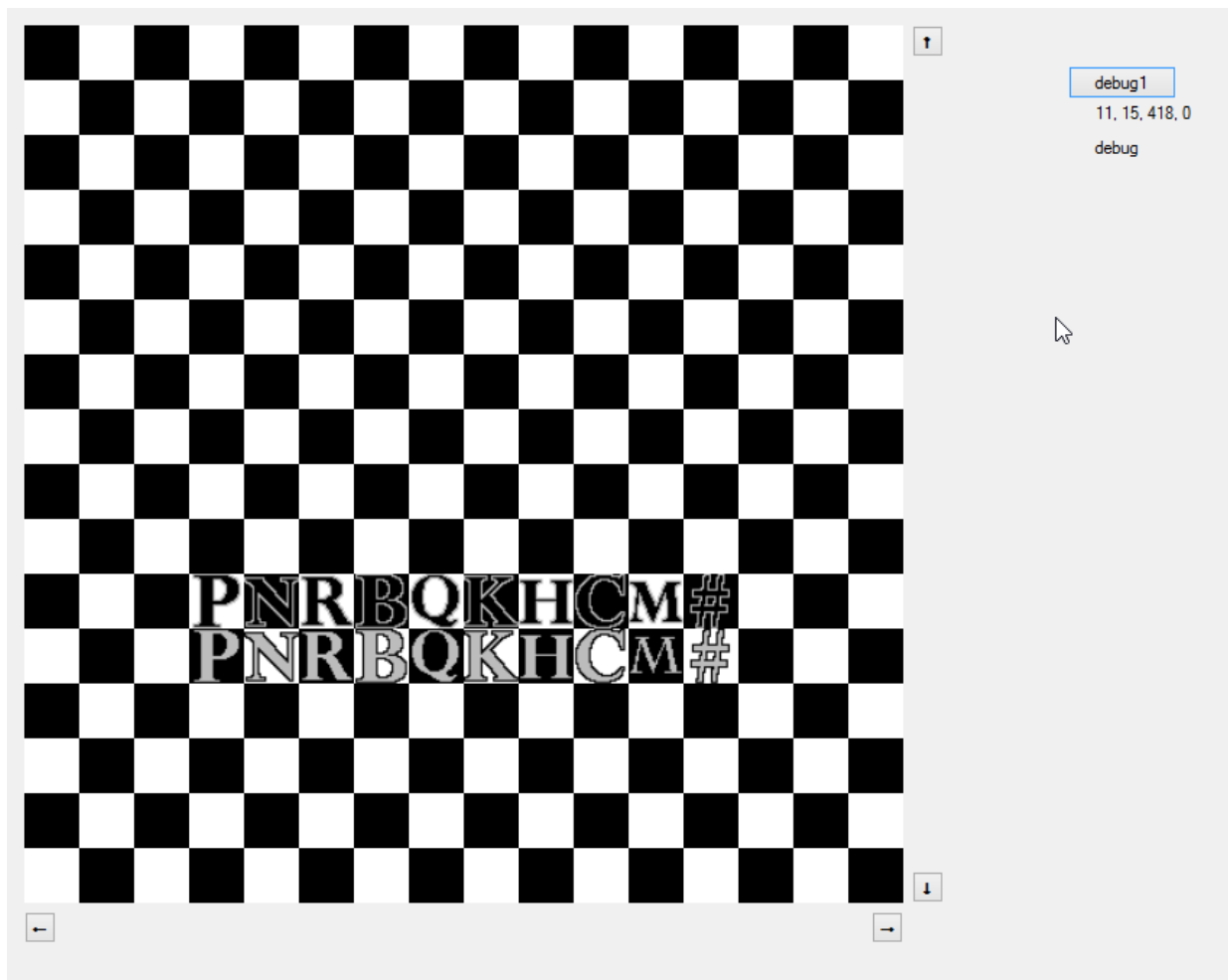


```
public static Square findSquareByIndex(int indexX, int indexY) {
    foreach (Square s in InfinteChess.board) {
        if (indexX == s.indexX && indexY == s.indexY) return s; }
    return null;
}
```

This only returns null if it couldn't match any of the squares in board. I decide to check the initialisation function for the game and I realise that the board is only being initialised after I press the button, but pieces is being initialised when the program loads. This means that every piece is being created with null squares, because there are no squares when the pieces are created. This is a simple fix, all I have to do is call `InitialiseBoard()` before pieces is initialised.

```
public InfinteChess()
{
    InitializeComponent();
    InitialiseBoard();
    pieces.AddRange(Piece.IntializePieces());
    bounds[1] = (size[0] - 1); bounds[3] = (size[1] - 1);
}
```

This now works as intended; some pieces have been drawn to the screen:



They don't look quite right though, having them the same size as the squares of the board doesn't look good. Changing their size to 32x32 looks like a more reasonable size, but they are now in the top left corner of each square, which is not what I want. Adding 3 to both the X and Y coordinates of each image fixes this issue, and their current appearance can be seen on the right.



While I was fixing the piece images, I noticed I use the value 38 (the size of the squares of the board) a lot in the code. I might want to change this value at a later point (perhaps for different resolutions), so I think it's a good idea now to create a variable which has the value of 38 and replace all instances of the number with the variable. I will call this variable *sf* for scale factor.

## Calculating Piece Movement

Having pieces is no good if they can't move. Making them move is trivial, but calculating where they can legally move will be more difficult.

There are two categories of pieces when it comes to how their movement is defined: static movement and linear movement. Which piece is which is defined in the table on the left. Static means that the squares that the piece can move to are finite and defined

STATIC	LINEAR
Pawn	Bishop
Mann	Rook
Knight	Chancellor
Hawk	Queen
King	

relative to the position of the piece. For example, a king can move one square in any direction, meaning it has a maximum of 8 possible moves at any time. Linear on the other hand means that the piece's movement is defined by a line or lines and can move in any number of squares in that line(s). In regular chess, this would

still mean they have a finite number of moves, since there are only a finite number of squares and the line of movement would hit the edge of the board quite soon. In infinite chess however, there is no edge; the line of movement will carry on indefinitely, and therefore so will the choice of moves. This means that the two categories will have to have their movement defined differently in terms of the code involved.

`calculateMovement` will be a function of the `Piece` class and take no arguments since it will calculate the movement of the instance of `Piece` that called it. The function will return a `List<Square>` which will represent the valid moves relative to the `Square` of the piece.

I will start with the easier of the two (static pieces). The function will start with the framework seen on the right.

```
public List<Square> calculateMovement() {
    List<Square> moves = Square.emptyList();
    switch (type) {
        case (PieceType.PAWN):
        case (PieceType.MANN):
        case (PieceType.KNIGHT):
        case (PieceType.HAWK):
        default:
            return moves;
    }
}
```

The pawn is the simplest piece; most of the time it just moves forward one Square (exceptions such as capturing, the first movement rule and en-passant can be added later).

```
case (PieceType.PAWN): {
    var direction = colour == PieceColour.WHITE ? square.indexY + 1 : square.indexY - 1;
    Square attempt = GameContainer.findSquareByIndex(square.indexX, direction);
    moves.Add(attempt);
    foreach (Piece p in Chess.pieces) { if (p.square == attempt) moves.Remove(attempt); }
    return moves;
}
```

The pawn is the only piece which has movement that depends on which colour it is, so I set up a variable to decide which direction movement should go. I then use attempt to represent the Square in front of the pawn in that direction and add it to the list. I then check if any other piece is on that Square, and if so, remove it from the list. This function will therefore output either one square in front of the pawn or no squares at all.

To test this, I could output the value of moves to one of the debug labels, but for other pieces it would become tedious to verify if the returned list is correct. An easier way would be to draw something on the squares returned so that I can visually see if it's working. This is something that will need to be done anyway so that the user can see what their valid moves are. I will write a basic function in GameContainer which will just draw a red rectangle on every Square that is returned, and then attach this to OnMouseClicked to draw a piece's movement when it is clicked.

```
public void drawMoves(Piece p) {
    Graphics g = CreateGraphics();
    foreach (Square s in p.calculateMovement()) {
        g.DrawRectangle(
            new Pen(Color.FromArgb(106, 255, 0, 0)),
            s.X, s.Y,
            Chess.sf, Chess.sf);
    }
    g.Dispose();
}
```

```
protected override void OnMouseClicked(MouseEventArgs e)
{
    Square cursorSquare = findSquareByCoords(e.X, e.Y);
    foreach (Piece p in Chess.pieces) {
        if (p.square == cursorSquare) drawMoves(p);
    }
}
```

Running this code and clicking on either pawn does the following:



Nothing drawn. However, since the pawns are in front of each other, this is actually the expected result. It is not possible to tell whether the code is working correctly in this situation, so I will move the rows of pieces apart. Doing this and then trying again:



There is a red square drawn where I expect, but it's a little hard to see, so I will make it thicker by just drawing additional smaller rectangles. After adjusting the constants to get the square centred properly, following is the code used to draw and the result of it:

```
public void drawMoves(Piece p) {
    Graphics g = CreateGraphics();
    foreach (Square s in p.calculateMovement()) {
        g.DrawRectangle(new Pen(Color.FromArgb(255, 255, 0, 0)), s.X, s.Y, Chess.sf-1, Chess.sf-1);
        g.DrawRectangle(new Pen(Color.FromArgb(255, 255, 0, 0)), s.X+1, s.Y+1, Chess.sf-3, Chess.sf-3);
        g.DrawRectangle(new Pen(Color.FromArgb(255, 255, 0, 0)), s.X+2, s.Y+2, Chess.sf-5, Chess.sf-5);
    }
    g.Dispose();
}
```



This result indicates that the code written so far is functioning correctly.

The next piece is the knight. This piece moves two squares in any orthogonal direction, and then an additional square in a direction perpendicular to the original movement (i.e. an L-shape). The best way to implement this movement will be to use an array to represent the movement, with each element being the offset in terms of squares for one possible move with respect to the origin (the square the piece is starting on). I will then iterate through each of these entries and check if there is a piece in any already. Any which are empty will be added to the list that will be returned.

```
case (PieceType.KNIGHT): {
    int[][] attempts = {
        new int[] {-2,1}, new int[] {-1,2},
        new int[] {2,1}, new int[] {1,2},
        new int[] {2,-1}, new int[] {1,-2},
        new int[] {-1,-2}, new int[] {-2,-1} };
    for (int i = 0; i < attempts.Length; i++) {
        Square s = GameContainer.findSquareByIndex(
            square.indexX + attempts[i][0],
            square.indexY + attempts[i][1]);
        bool valid = true;
        foreach (Piece p in Chess.pieces) {
            if (p.square == s) { valid = false; }
        }
        if (valid) { moves.Add(s); }
    }
    return moves;
}
```

This is not too different from the code for the pawn. The result of clicking the black knight can be seen on the left, which is the expected result; 6 of the 8 possible squares are free, while the other 2 are blocked by a white pawn and white rook.



The next piece is the king, which also shares its movement with the man. Even though the king has a special status among pieces, right now I am only concerned with the movement, so there should be no issues with using the exact same code for both.

For these two pieces, I can simply copy the code for the knight, but with the numbers in the array adjusted:

```

case (PieceType.KING): {
    int[][] attempts = {
        new int[] { -1, -1 }, new int[] { -1, 0 },
        new int[] { -1, 1 }, new int[] { 0, -1 },
        new int[] { 0, 1 }, new int[] { 1, -1 },
        new int[] { 1, 0 }, new int[] { 1, 1 } };
    for (int i = 0; i < attempts.Length; i++)
    {
        Square s = GameContainer.findSquareByIndex(
            square.indexX + attempts[i][0],
            square.indexY + attempts[i][1]);
        bool valid = true;
        foreach (Piece p in Chess.pieces)
        {
            if (p.square == s) { valid = false; }
        }
        if (valid) { moves.Add(s); }
    }
    return moves;
}
case (PieceType.MANN): { goto case PieceType.KING; }

```

The final static piece is the hawk. This piece moves either 2 or 3 squares in any direction and can jump over pieces in the way. This means the code would once again be exactly the same, but with a modified array. Having all this repeated code seems rather redundant, so I decide to try and have each of these pieces direct to the same function. The most readable way to do this is to store the arrays used previously in text files as numbers that can be read and interpreted in the same way. The file for the king can be seen to the right; as you can see it contains all the same numbers as the array did, but it makes it much easier to deal with and the array is no longer hardcoded into the game. The modified code with the file read operation can be seen below.

```

-1,1
-1,0
-1,-1
0,1
0,-1
1,1
1,0
1,-1

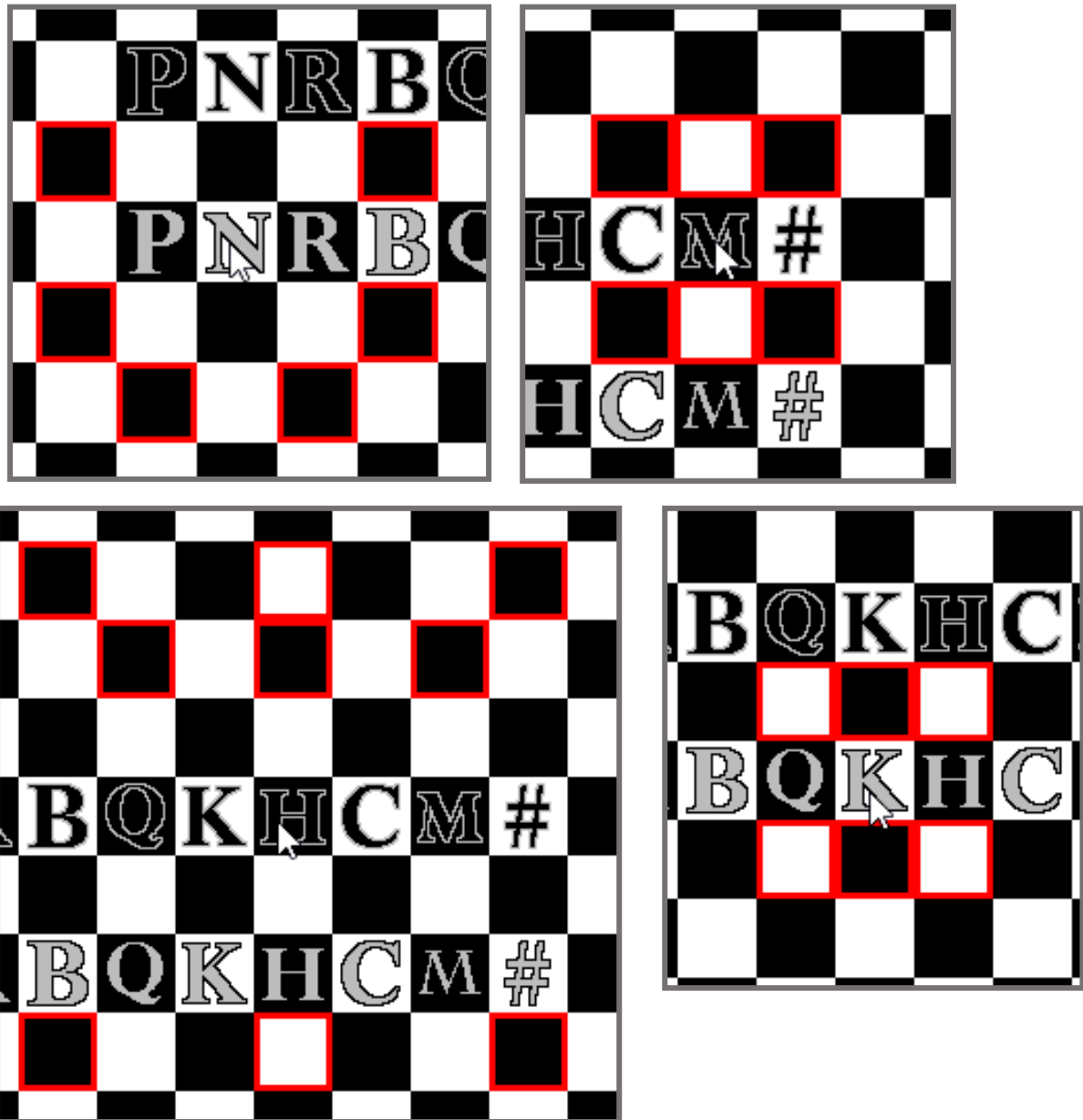
```

```

case (PieceType.KING): {
    string[] attempts = File.ReadAllLines($"res/movement/{type.ToString()}.txt");
    foreach (string att in attempts)
    {
        Square s = GameContainer.findSquareByIndex(
            square.indexX + Int32.Parse(att.Split(',')[0]),
            square.indexY + Int32.Parse(att.Split(',')[1]));
        bool valid = true;
        foreach (Piece p in Chess.pieces) {
            if (p.square == s) { valid = false; }
        }
        if (valid) moves.Add(s);
    }
    return moves;
}
case (PieceType.MANN): { goto case PieceType.KING; }
case (PieceType.KNIGHT): { goto case PieceType.KING; }
case (PieceType.HAWK): { goto case PieceType.KING; }

```

And the results of clicking these pieces:



These appear to be functioning as expected. By moving pieces out from the rows, I can verify each of the possible moves is being checked as intended. This means the movement calculation for static pieces is complete.

Now for the linear pieces. As mentioned earlier, I will have to use a different approach for these pieces, because they don't have a set of predefined squares they can move to. As well as this, they cannot jump over pieces like the hawk, so they must account for other pieces stopping a line of movement at any point.



The algorithm for these kind of pieces is going to be something like the following:

```
case Bishop: {
    create empty list of squares
    int max = largest distance from the piece to the edge of visible board
    for (int i = 1; i <= j; i++) {
        Square attempt = findSquare(this.indexX - i, this.indexY - i)
        if there's a piece on the square, break out of the loop
        add attempt to list
    }
    for (int i = 1; i <= j; i++) {
        Square attempt = findSquare(this.indexX - i, this.indexY + i)
        if there's a piece on the square, break out of the loop
        add attempt to list
    }
    //2 more loops for the other 2 directions
    return list
}
```

There is a built-in function to find the largest of two integers, but I need to find the

```
public int findLargest(int[] i) {
    int j = int.MinValue;
    foreach (int k in i) { if (k > j) j = k; }
    return j;
}
```

largest of four in this case. I may need to do this again at another point in development, so I will write a function that will take an array of integers and return the largest one.

The initial iteration of this code looks like this:

```
case (PieceType.BISHOP): {
    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0],
        square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX,
        Chess.bounds[3]-square.indexY });
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY - i);
        foreach (Piece p in Chess.pieces) { if (p.square == attempt) break; }
        moves.Add(attempt);
    }
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX + i, square.indexY - i);
        foreach (Piece p in Chess.pieces) { if (p.square == attempt) break; }
        moves.Add(attempt);
    }
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY + i);
        foreach (Piece p in Chess.pieces) { if (p.square == attempt) break; }
        moves.Add(attempt);
    }
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX + i, square.indexY + i);
        foreach (Piece p in Chess.pieces) { if (p.square == attempt) break; }
        moves.Add(attempt);
    }
    return moves;
}
```



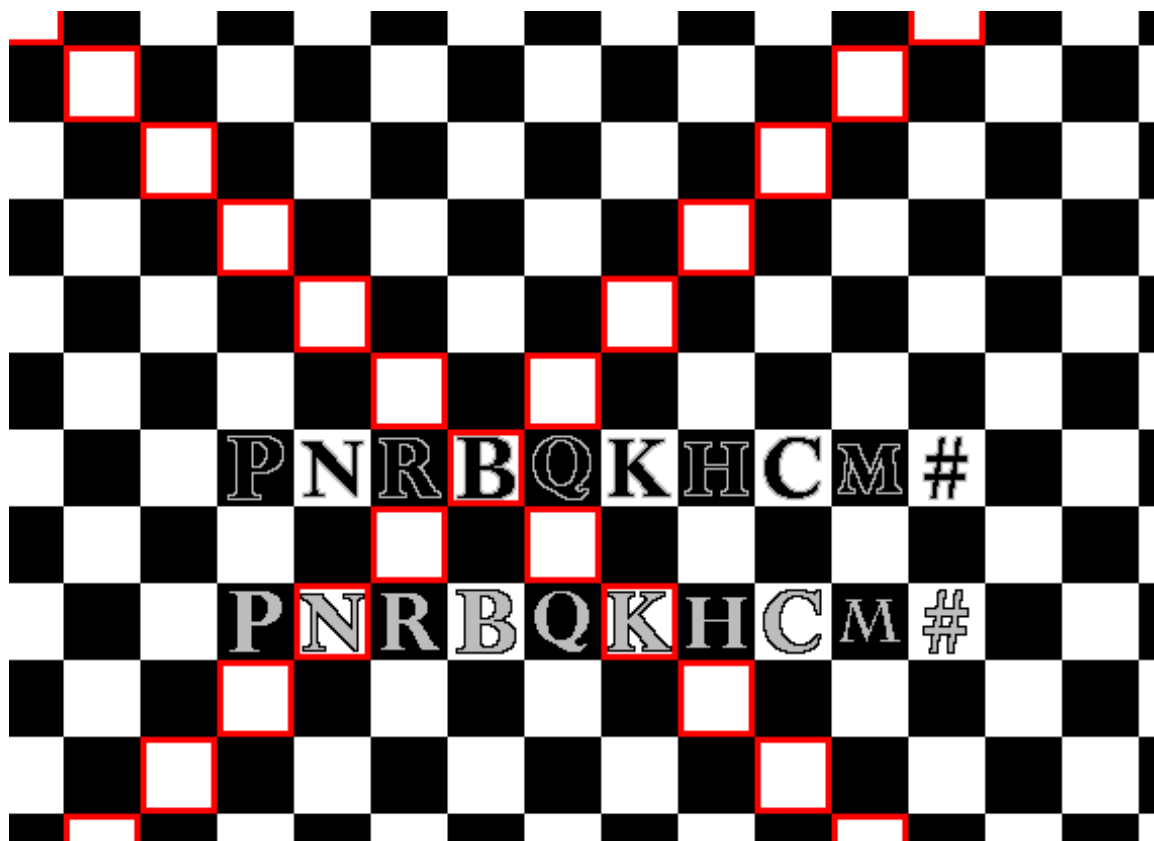
However, this throws a `NullPointerException` when a bishop is clicked. The limit I am using for each for loop is the largest of the 4 distances from the piece to the edges of the board. However, this means that in some direction the program is attempting to check past the edge of the visible board if it's not stopped by another piece being in the way, which in a lot of cases will lead it to check squares which don't exist, and `findSquareByIndex` will return `null` in these cases. This means that `drawMovement` throws an exception (cannot draw a null square).

To fix this, I need to break out of the loop if `attempt` becomes equal to `null`. The standard approach to this would be to just add an `if` statement with a `break` statement in each loop, but this code is already a bit messy, and that would make it worse. A more clever and much shorter way to fix this is to use a null-conditional operator in the following way:

```
Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY - i) ?? square;
```

The program will check for the next `Square` in the line as usual, but if the function returns `null`, instead of setting `attempt` to be `null`, it instead sets it to whatever is to the right of `??` (the null-conditional operator), which in this case is `square`. `square` refers to the square of the piece which called this function (i.e. the square the bishop is on), and therefore necessarily will have a piece on it. This means that in the `foreach` loop, this square will be caught, and the `break` statement will be reached, ending the loop for that line.

This is the result of clicking a bishop with the new adjustments:



The program hasn't crashed, which is good, but it is clearly not playing by the rules here. The bishop calculations are ignoring pieces that are in the way. The reason for this is a simple oversight I made; `break` only takes execution out of one loop, and not more. This means that in my code, the `break` statements are jumping out of the `foreach` loop, but then not out the enclosing `for` loop, so the move is getting added to the list anyway.

The best way to fix this is to have a `bool` defined at the start of the entire case which is set to `true` initially, and then becomes `false` when a piece is found within a `foreach`.

There is then an `if` statement after the `foreach` which will add the move if the `bool` is

```
public static bool checkSquareForPiece(Square s) {
    foreach (Piece p in pieces) {
        if (p.square == s) return true;
    }
    return false;
}
```

`true`, and `break` if `false`. Looking at the code for this class in general, I have used this same structure (`foreach`) many times with no changes at all. I think it would be useful here to

move this to another function which will return either `true` if there is a piece on the square, or `false` otherwise to reduce the clutter of this class. This can be seen on the right.

This new iteration looks like this:

```
case (PieceType.BISHOP): {
    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0],
        square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX,
        Chess.bounds[3]-square.indexY } );
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY - i) ?? square;
        if (Chess.checkSquareForPiece(attempt)) break;
        moves.Add(attempt);
    }
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX + i, square.indexY - i) ?? square;
        if (Chess.checkSquareForPiece(attempt)) break;
        moves.Add(attempt);
    }
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY + i) ?? square;
        if (Chess.checkSquareForPiece(attempt)) break;
        moves.Add(attempt);
    }
    for (int i = 1; i <= max; i++) {
        Square attempt = GameContainer.findSquareByIndex(square.indexX + i, square.indexY + i) ?? square;
        if (Chess.checkSquareForPiece(attempt)) break;
        moves.Add(attempt);
    }
    return moves;
}
```

I have also gone back to the previous piece code and replaced any similar `foreach` loops with a call of `checkSquareForPiece`.

The result of this iteration:



Much better. However, this code can be improved further. Currently I have 4 consecutive for loops which all share 2 lines, and the third line only differs in two characters. I think these can be combined into just one loop. If I just copied each three line section into one for loop, they would all stop as soon as one stopped because of the break statement. I need a way to track each three line section and have it not execute if necessary.

To do this, I will use a `bool[]` which contains 4 values, one for each direction. They all initially started as false, and every time a square is checked, the value for that line is set to whatever `checkSquareForPiece` returns. Additionally, each three line segment only executes if the tracker value for that section is false. What this means is that as soon as something stops one of the lines, just that line will stop and the rest will continue, until all four have stopped (i.e. all four values are true).

While this will make the code a bit messy, it will computationally be a lot faster, since I am now only doing the main iteration once instead of four times.

The implementation of this looks like this:

```
case (PieceType.BISHOP): {
    bool[] tracker = { false, false, false, false };
    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0],
        square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX,
        Chess.bounds[3]-square.indexY } );
    for (int i = 1; i <= max; i++) {
        if (!tracker[0]) {
            Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY - i) ?? square;
            tracker[0] = Chess.checkSquareForPiece(attempt);
            if (!tracker[0]) moves.Add(attempt);
        }
        if (!tracker[1]) {
            Square attempt = GameContainer.findSquareByIndex(square.indexX + i, square.indexY - i) ?? square;
            tracker[1] = Chess.checkSquareForPiece(attempt);
            if (!tracker[1]) moves.Add(attempt);
        }
        if (!tracker[2]) {
            Square attempt = GameContainer.findSquareByIndex(square.indexX - i, square.indexY + i) ?? square;
            tracker[2] = Chess.checkSquareForPiece(attempt);
            if (!tracker[2]) moves.Add(attempt);
        }
        if (!tracker[3]) {
            Square attempt = GameContainer.findSquareByIndex(square.indexX + i, square.indexY + i) ?? square;
            tracker[3] = Chess.checkSquareForPiece(attempt);
            if (!tracker[3]) moves.Add(attempt);
        }

        if (tracker == new bool[] { true, true, true, true }) { break; }
    }
    return moves;
}
```

This gives the same result as the previous iteration. While I am now calculating the movement 4 times faster, there is still a lot of repeated code in this function. Using another array which will hold information on which direction to check (which coordinates should be subtracted from in `findSquareByIndex`), I can make this even better:

```
case (PieceType.BISHOP): {
    bool[] tracker = { false, false, false, false };
    int[][] direction = { new int[]{-1,-1}, new int[]{-1,1}, new int[]{1,-1}, new int[]{1,1} };

    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0], square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX, Chess.bounds[3]-square.indexY } );

    for (int i = 1; i <= max; i++) {
        for (int j = 0; j < 4; j++) {
            if (!tracker[j]) {
                Square attempt = GameContainer.findSquareByIndex(
                    square.indexX - i*direction[j][0], square.indexY - i*direction[j][1]) ?? square;
                tracker[j] = Chess.checkSquareForPiece(attempt);
                if (!tracker[j]) moves.Add(attempt);
            }
        }
    }
    return moves;
}
```

This new code is much shorter, and still gives the same result. This will be the final iteration for the bishop.

The rook is pretty much identical to the bishop, except the lines are orthogonal instead of diagonal. This means the code for the bishop can be copied, direction can be modified, and it will work:

```
case (PieceType.ROOK): {
    bool[] tracker = { false, false, false, false };
    int[][] direction = { new int[]{-1,0}, new int[]{1,0}, new int[]{0,-1}, new int[]{0,1} };
    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0], square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX, Chess.bounds[3]-square.indexY } );
    for (int i = 1; i <= max; i++) {
        for (int j = 0; j < 4; j++) {
            if (!tracker[j]) {
                Square attempt = GameContainer.findSquareByIndex(
                    square.indexX - i*direction[j][0], square.indexY - i*direction[j][1]) ?? square;
                tracker[j] = Chess.checkSquareForPiece(attempt);
                if (!tracker[j]) moves.Add(attempt);
            }
        }
    }
    return moves;
}
```

The result of this code can be seen to the right. However, once again, there is a lot of redundant code here. Only one line differs between the code for the bishop and the rook. This means I could easily direct the rook to run the code for the bishop, and just adjust the constants on the fly as necessary. The improved code can be seen below.

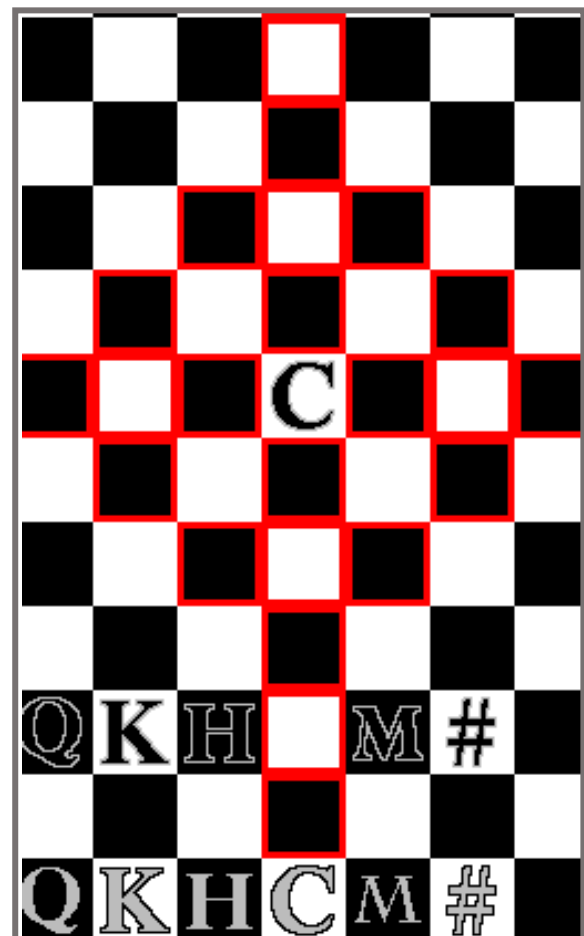


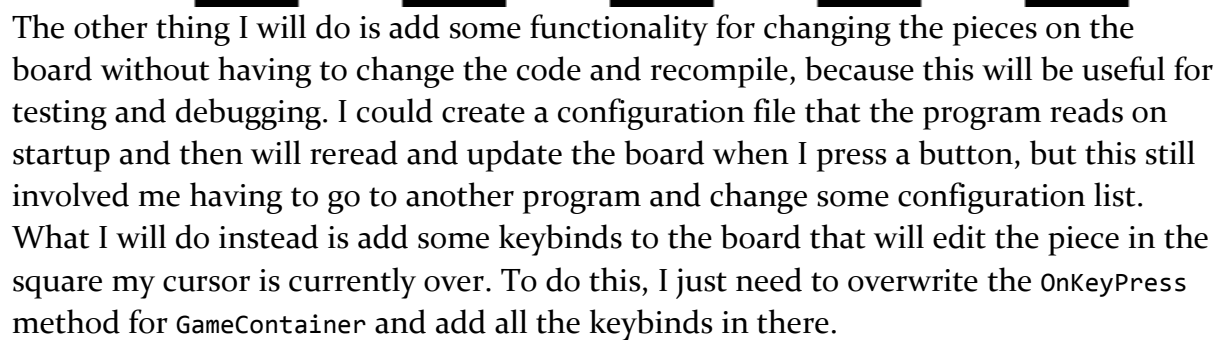
```
case (PieceType.ROOK): { goto case PieceType.BISHOP; }
case (PieceType.BISHOP): {
    bool[] tracker = { false, false, false, false };
    int[][] direction = type == PieceType.BISHOP ?
        new int[][] { new int[]{-1,-1}, new int[]{-1,1}, new int[]{1,-1}, new int[]{1,1} } :
        new int[][] { new int[]{-1,0}, new int[]{1,0}, new int[]{0,-1}, new int[]{0,1} };
    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0], square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX, Chess.bounds[3]-square.indexY } );
    for (int i = 1; i <= max; i++) {
        for (int j = 0; j < 4; j++) {
            if (!tracker[j]) {
                Square attempt = GameContainer.findSquareByIndex(
                    square.indexX - i*direction[j][0], square.indexY - i*direction[j][1]) ?? square;
                tracker[j] = Chess.checkSquareForPiece(attempt);
                if (!tracker[j]) moves.Add(attempt);
            }
        }
    }
    return moves;
}
```

The two remaining pieces are the queen and chancellor. These will be simple to complete, because their movement is a combination of the movements of other pieces. A queen is a combination of the rook and the bishop, while the chancellor is a combination of the knight and the rook.

```
case PieceType.QUEEN: {
    moves.AddRange(new Piece(
        PieceType.BISHOP, square, PieceColour.WHITE).calculateMovement());
    moves.AddRange(new Piece(
        PieceType.ROOK, square, PieceColour.WHITE).calculateMovement());
    return moves;
}
case PieceType.CHANCELLOR: { //KNIGHT + ROOK
    moves.AddRange(new Piece(
        PieceType.KNIGHT, square, PieceColour.WHITE).calculateMovement());
    moves.AddRange(new Piece(
        PieceType.ROOK, square, PieceColour.WHITE).calculateMovement());
    return moves;
}
```

And their movements display as follows:





```
protected override void OnKeyPress(KeyPressEventArgs e)
{
    Label l = (Label)Parent.Controls.Find("debug3", false)[0];
    Button b = (Button)Parent.Controls.Find("begin", false)[0];

    Point a = Parent.PointToClient(new Point(Cursor.Position.X, Cursor.Position.Y));
    Square cursorSquare = findSquareByCoords(a.X - Location.X, a.Y - Location.Y);
    Piece target = null;
    foreach (Piece p in Chess.pieces) { if (p.square == cursorSquare) { target = p; } }

    if (cursorSquare != null) {
        if (target != null) {
            if (e.KeyChar == '\b') Chess.pieces.Remove(target);
            if (e.KeyChar == 'c') target.altColour();
        }
        else {
            switch (e.KeyChar) {
                case '0': {
                    Chess.pieces.Add(new Piece(PieceType.NONE, cursorSquare, PieceColour.WHITE));
                    break; }
                case '1': {
                    Chess.pieces.Add(new Piece(PieceType.PAWN, cursorSquare, PieceColour.WHITE));
                    break; }
                case '2': {
                    Chess.pieces.Add(new Piece(PieceType.BISHOP, cursorSquare, PieceColour.WHITE));
                    break; }
                case '3': {
                    Chess.pieces.Add(new Piece(PieceType.KNIGHT, cursorSquare, PieceColour.WHITE));
                    break; }
                case '4': {
                    Chess.pieces.Add(new Piece(PieceType.HAWK, cursorSquare, PieceColour.WHITE));
                    break; }
                case '5': {
                    Chess.pieces.Add(new Piece(PieceType.HAWK, cursorSquare, PieceColour.WHITE));
                    break; }
                case '6': {
                    Chess.pieces.Add(new Piece(PieceType.ROOK, cursorSquare, PieceColour.WHITE));
                    break; }
                case '7': {
                    Chess.pieces.Add(new Piece(PieceType.CHANCELLOR, cursorSquare, PieceColour.WHITE));
                    break; }
                case '8': {
                    Chess.pieces.Add(new Piece(PieceType.QUEEN, cursorSquare, PieceColour.WHITE));
                    break; }
                case '9': {
                    Chess.pieces.Add(new Piece(PieceType.KING, cursorSquare, PieceColour.WHITE));
                    break; }
            }
        }
    }
    b.PerformClick();
}
}
```

This code allows me to create new pieces by using the number keys, delete a piece using backspace, and alternate the colour of a piece using c.

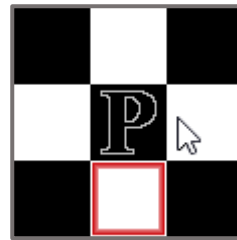
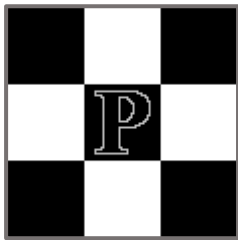


## Piece Movement Tests

Now I can begin testing the functionality of every piece. I will do a series of tests; for each, I will outline a starting configuration, explain the steps I will take, the expected outcome of those steps, and the actual outcome for those steps.

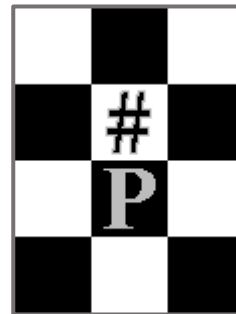
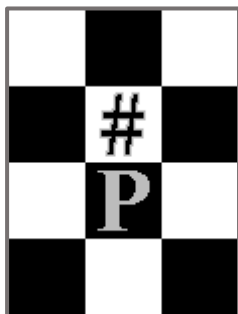
### *Lone Pawn*

- \* The black pawn will be clicked
- \* The expected outcome is the square directly below the pawn becomes highlighted.
- \* The actual outcome was as expected.



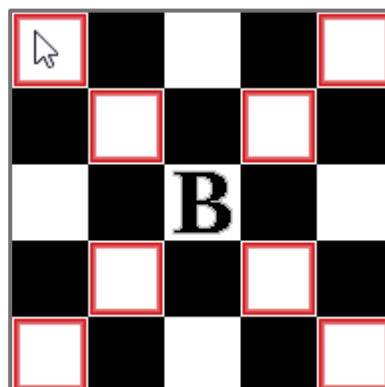
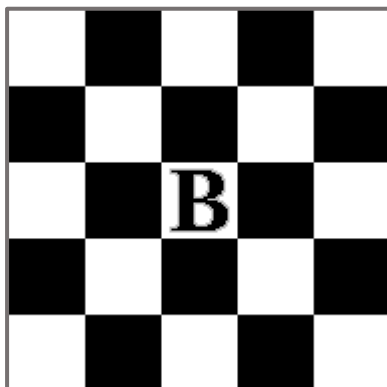
### *Blocked Pawn*

- \* The white pawn will be clicked.
- \* The expected outcome is no squares will become highlighted.
- \* The outcome was as expected



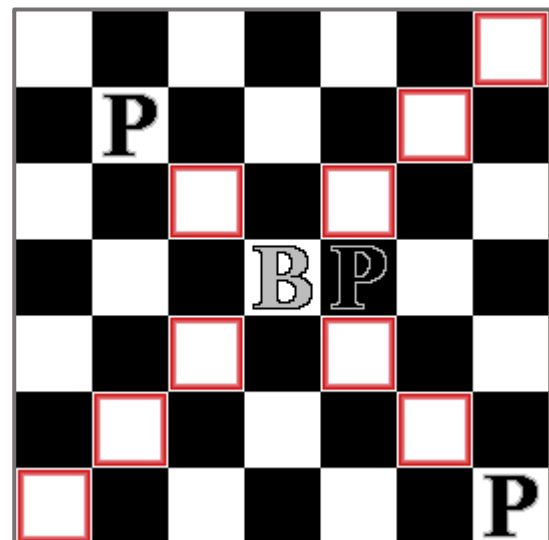
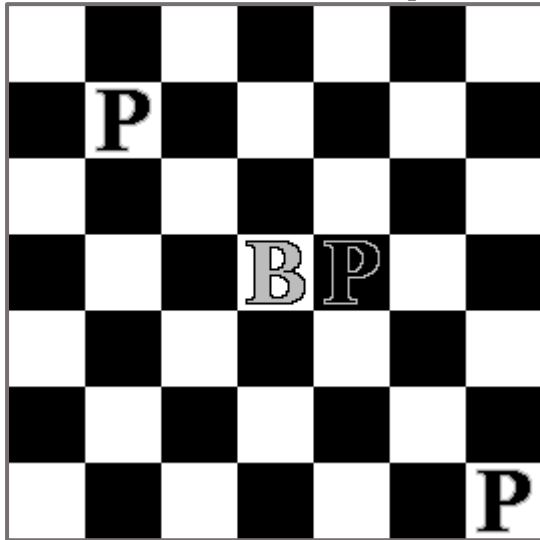
### *Lone Bishop*

- \* The black bishop will be clicked.
- \* The squares which are diagonal from the bishop should become highlighted.
- \* The outcome was as expected.



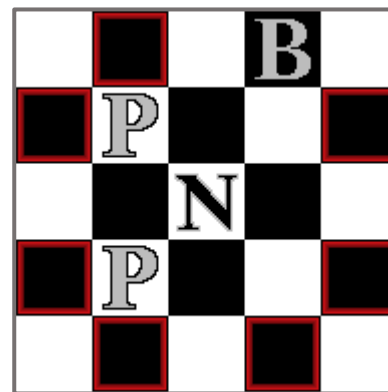
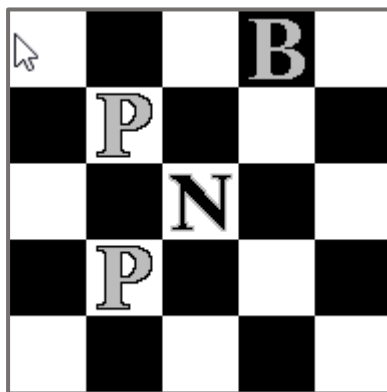
### *Doubly Blocked Bishop*

- \* The white bishop will be clicked.
- \* All squares on the top right and bottom left diagonals should become highlighted
- \* One square on the top left diagonal and two squares on the bottom right diagonal should become highlighted before being blocked by the pawns
- \* The outcome was as expected.



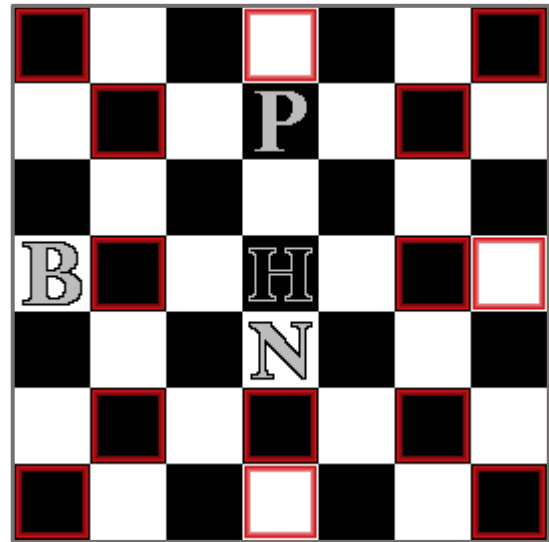
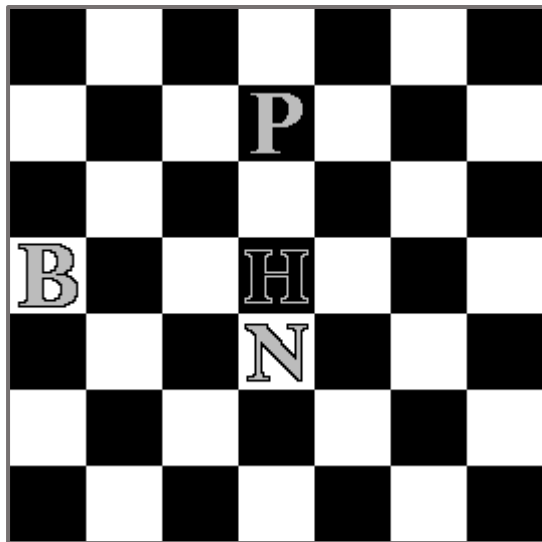
### *Blocked Knight*

- \* The black knight will be clicked.
- \* All squares which are adjacent to the corners of the square shown apart from the one with a bishop in it should become highlighted.
- \* The outcome was as expected.



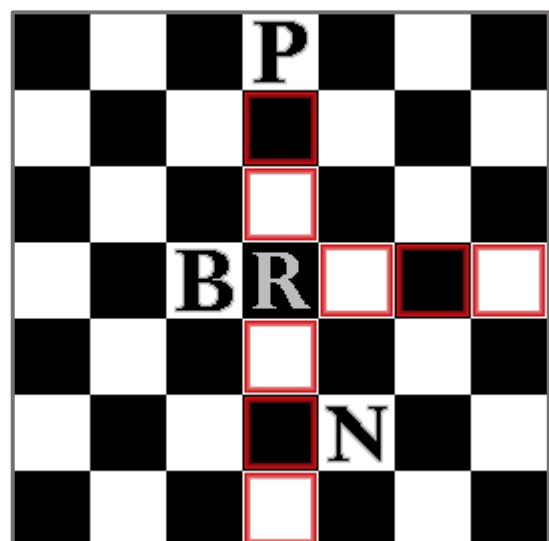
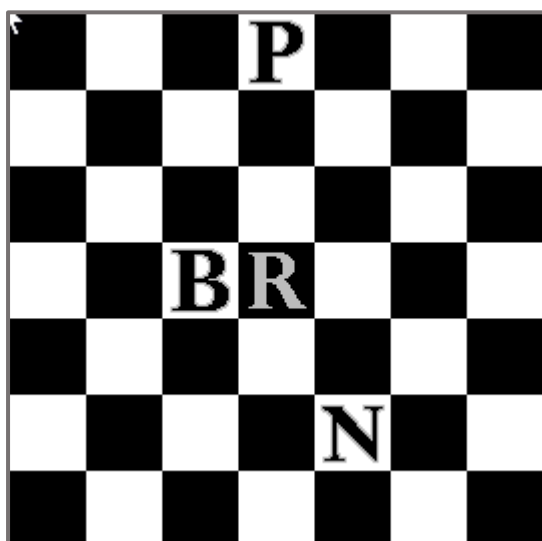
### *Doubly Blocked Hawk*

- \* The black hawk will be clicked.
- \* All squares which are two or three squares away from the hawk in any orthogonal or diagonal direction except the pieces occupied by the pawn and bishop should become highlighted.
- \* The outcome was as expected.



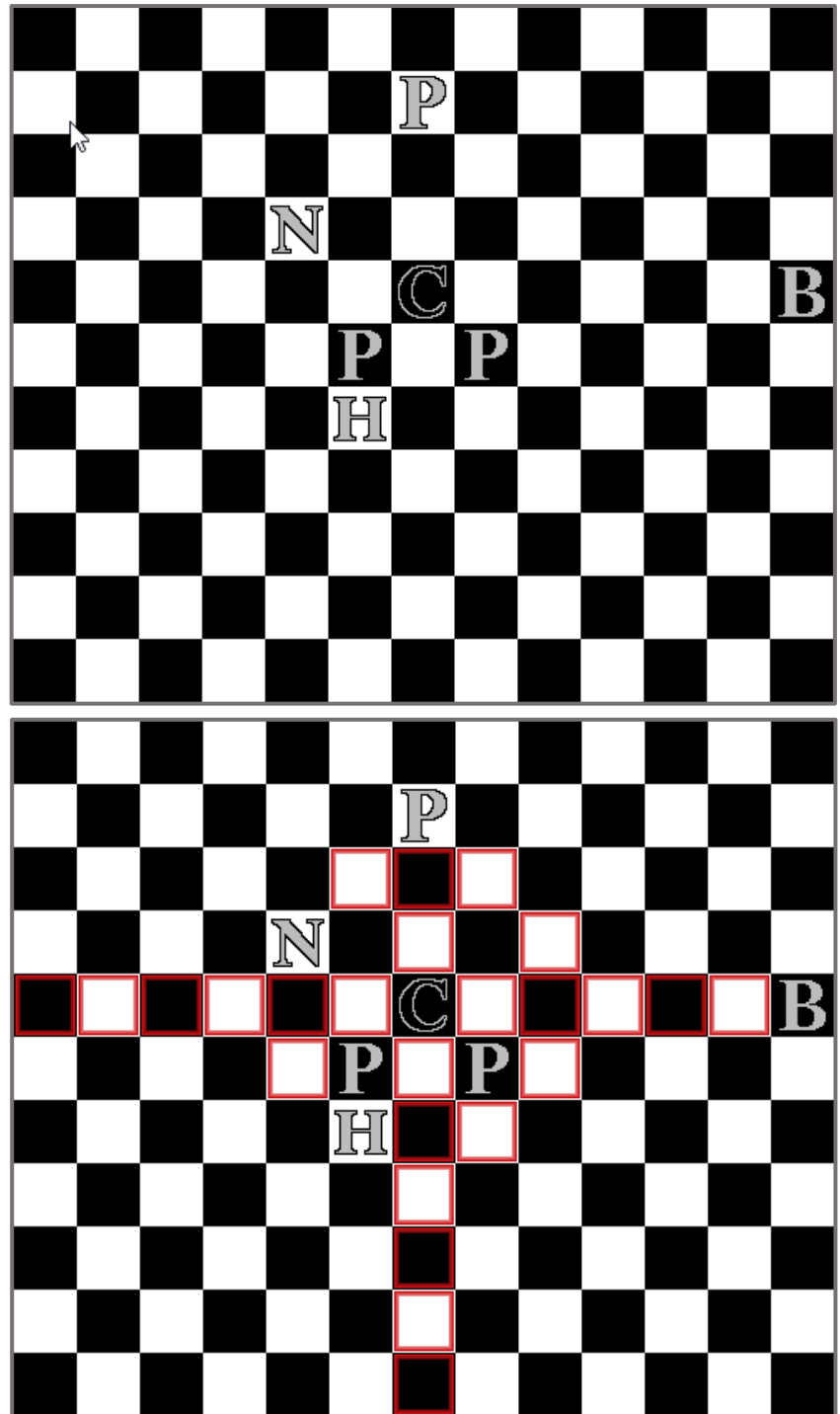
### *Doubly Blocked Rook*

- \* The white rook will be clicked
- \* All squares to the right and below the rook should become highlighted
- \* Two squares above the rook should become highlighted, before being blocked by the pawn
- \* No squares to the left should be highlighted, since they are blocked by the bishop.
- \* The outcome was as expected.



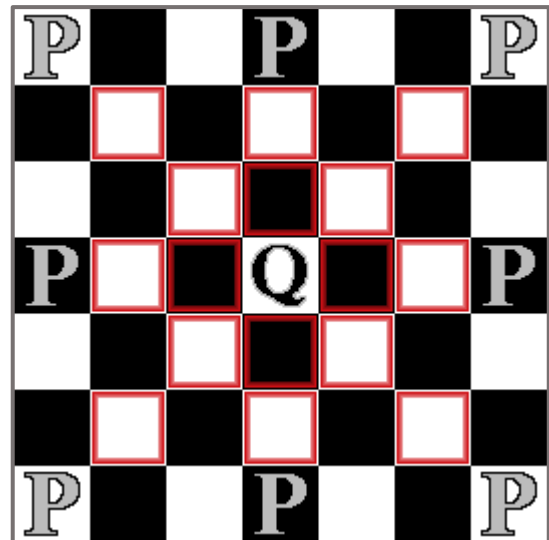
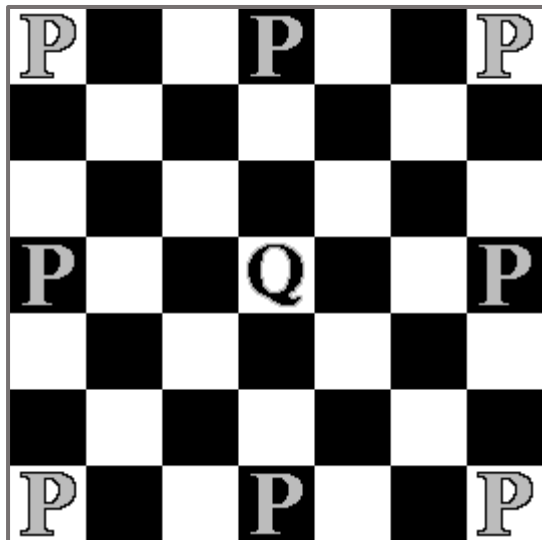
### *Blocked Chancellor*

- \* The black chancellor will be clicked.
- \* All squares to the left and below the chancellor should become highlighted.
- \* Two squares above and five squares to the right should be highlighted before being blocked by the pawn and bishop, respectively.
- \* The squares which represent the movement of a knight should be highlighted, except for the one to the left blocked by the knight, and below blocked by the hawk.
- \* The outcome was as expected.



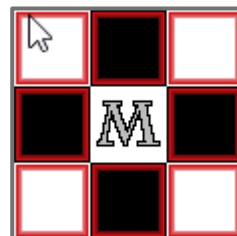
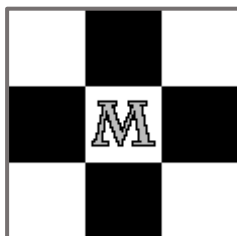
### *Blocked Queen*

- \* The black queen will be clicked.
- \* Two squares in each orthogonal and diagonal direction should become highlighted, before being blocking by the pawns.
- \* The outcome was as expected.



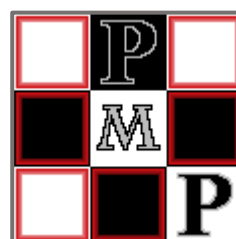
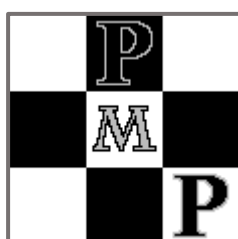
### *Lone Mann*

- \* The black mann will be clicked.
- \* All adjacent squares should become highlighted.
- \* The outcome was as expected.



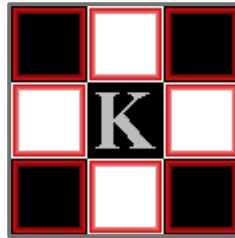
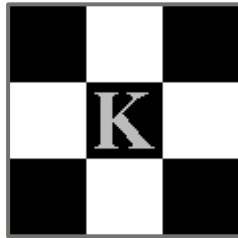
### *Blocked Mann*

- \* The white mann will be clicked.
- \* All adjacent squares except the ones above and to the bottom right should become highlighted.
- \* The outcome was as expected.



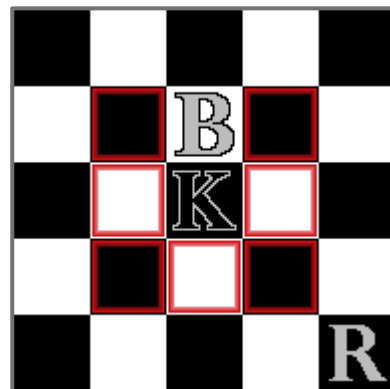
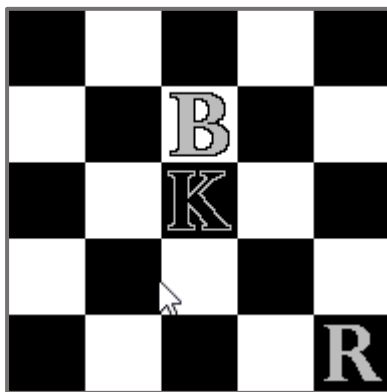
### *Lone King*

- \* The white king will be clicked.
- \* All adjacent squares should become highlighted.
- \* The outcome was as expected.



### *Blocked King*

- \* The black king will be clicked.
- \* All adjacent squares except the one above blocked by the bishop should become highlighted.
- \* The outcome was as expected.

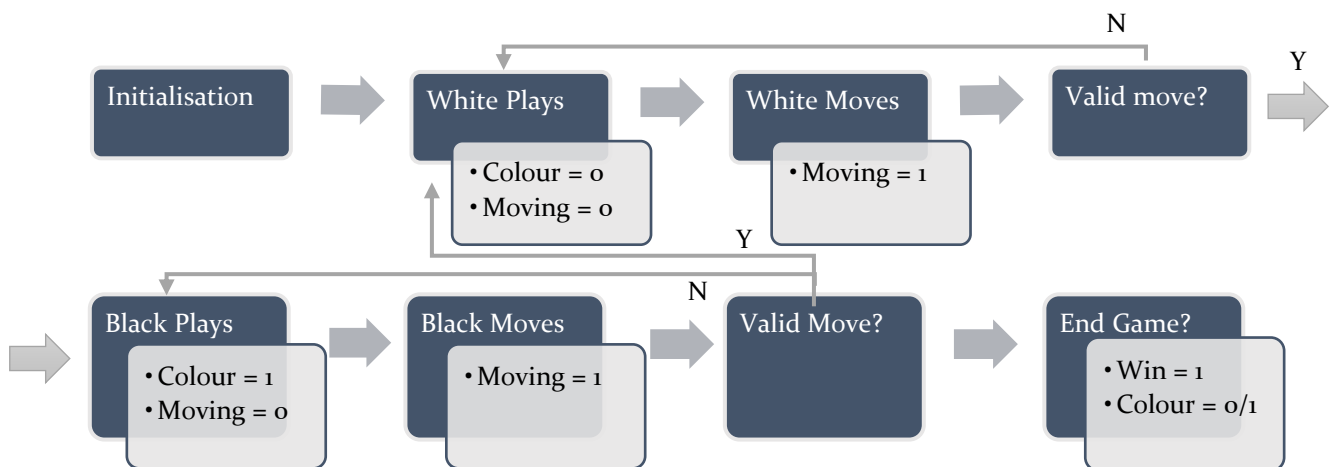


I can conclude, given the results of these tests, that the movement of all pieces is being properly calculated.

## Turn Framework and Game States

Now that I know where pieces should move, the next logical thing to implement is making them actually move. Piece moving is a part of the turn-based system of the game (making sure the player can only move during their turn, and consequently that they can only move their pieces in their turn), and this requires some way to keep track of the current state of the game (i.e. who's turn it is). Since these things are all part of the same idea, they will all be implemented in this section.

The idea for the flow of the game is simple enough, and is illustrated in the diagram below; stages are named in the dark boxes, and the flow of the game is represented by arrows. The lighter boxes represent internal flags which will be set at each point so that the game can know what is happening at any point.



The enumerations I have been using previously are those in which only one value can be used at once (i.e. a piece can only be one type of piece). If I used this for the game states, I would need to create a separate entry in the enum for each combination of multiple factors. Instead, I can use a different type of enumeration which allows me to use combinational flags to describe the state. These can be applied in this situation because the state of the game can be described with a number of binary variables.

The definition of this enum looks like this:

```
[Flags] public enum State {
    Colour = 0x01, Move = 0x02, Check = 0x04, Win = 0x08, Stale = 0x0F
}
```

Each variable is represented by a single binary bit, so the entire state of the game is described with a 5-bit binary number. For example, if white has just caused a stalemate after putting black in check, the game would be described by 00101<sub>2</sub>.

The useful thing about having these values represented as a binary number is that it simplifies getting and setting the state. For example, the code that decides whether to draw the movement of a piece or move a piece might look like the following if I was not using flags:

```
public void handleTurn() {
    if (state == GameState.PLAY_BLACK || state == GameState.PLAY_WHITE) {
        //find piece and draw its movement
        if (state == GameState.PLAY_BLACK) state = GameState.MOVING_BLACK;
        else state = GameState.MOVING_WHITE;
    }
    else if (state == GameState.MOVING_BLACK || state == GameState.MOVING_WHITE){
        //move piece
        if (state == GameState.MOVING_BLACK) state = GameState.PLAY_BLACK;
        else state = GameState.PLAY_WHITE;
    }
}
```

Using flags allows me to do binary arithmetic to toggle these values easily:

```
public void handleTurn() {
    if (!state.HasFlag(GameState.MOVING)) {
        //find piece and draw its movement
        state ^= GameState.MOVING;
    }
    else {
        //move piece
        state ^= (GameState.MOVING | GameState.COLOUR);
    }
}
```

Where ^ is the bitwise XOR operator and | is the bitwise AND operator.

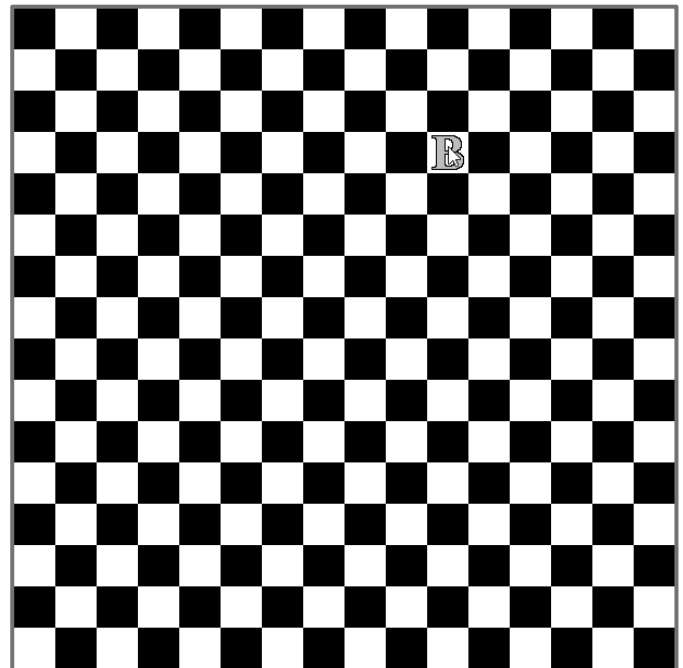
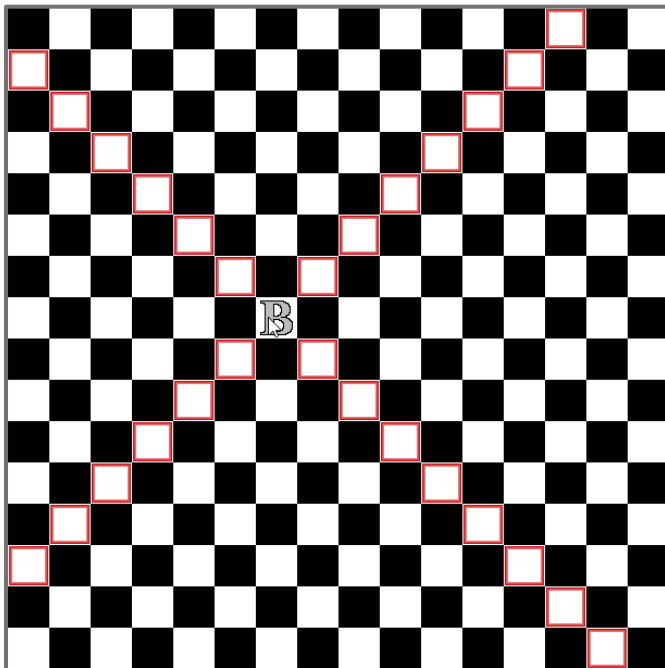
When the game starts, state will have a value of 0x0, meaning it is white's turn, there is no piece currently moving, no one is in check, neither player has won, and there is no stalemate.

Turns will be handled by a function, `handleTurn`, which will be in `GameContainer`, and is called when a mouse click occurs somewhere on the board. There are two arguments, Piece `p` and Square `s`, that are passed into `handleTurn` by `OnMouseClicked`. These arguments represent the piece that was under the mouse cursor when the click occurred (which may be null), and the square the cursor was over. Implementing the flowchart and algorithm gives the following:

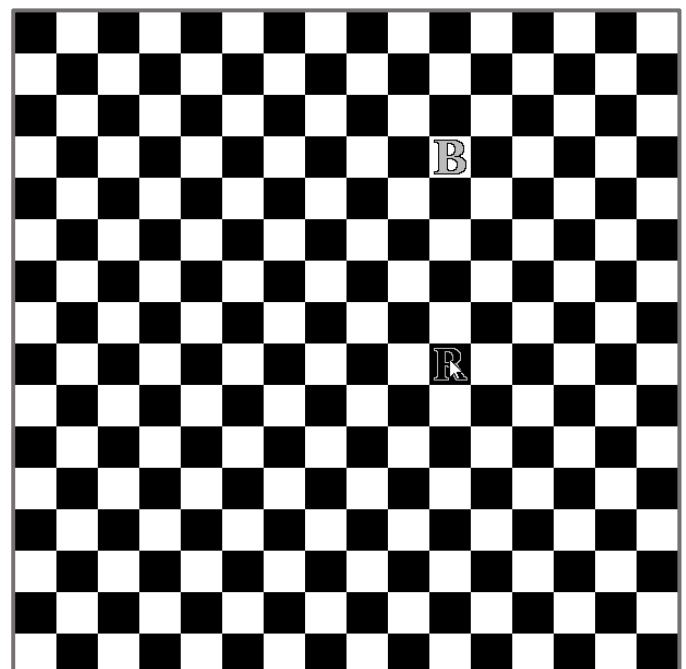
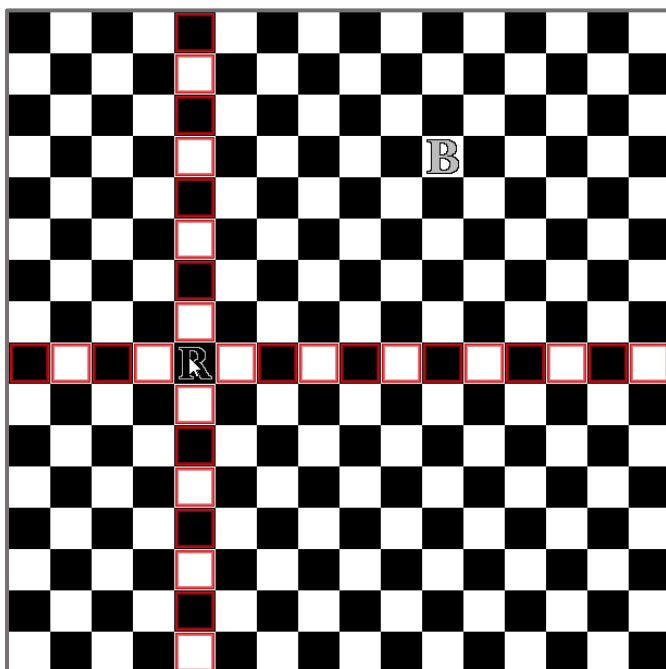
```
public void handleTurn(Piece p, Square s) {
    if (!state.HasFlag(GameState.MOVE)) {
        if (p != null && !state.HasFlag(GameState.COLOUR) == p.colour.HasFlag(PieceColour.WHITE)) {
            pieceMoving = p;
            drawMoves(p);
            state ^= GameState.MOVE;
        }
    }
    else {
        if (pieceMoving.calculateMovement().Contains(s)) {
            pieceMoving.move(s);
            state ^= (GameState.MOVE | GameState.COLOUR);
        }
        else state ^= GameState.MOVE;
        c.drawBoard();
        pieceMoving = null;
    }
}
```



To test it out, I start the game and create a bishop. Since white moves first, the bishop must be white. Clicking it drew the possible moves for the bishop, and then clicking one of the highlighted squares moved the bishop to that square. Clicking the bishop again afterwards did nothing, since it is now black's turn.



I create a black rook and click it. This causes the rook movement to become highlighted. I click one of the highlighted squares again and the rook moves.



If after clicking a piece to show the moves I click a non-highlighted square, the highlighted squares disappear but I can click the piece again. A turn does not end until a piece moves.

## Capturing Pieces

Capturing pieces is of course an integral part of chess, and since pieces can move and turns are implemented, this seems like the next logical step.

I will need to re-write some of the movement calculation code, because currently a square which has another piece on will never be returned as a valid move. However, to be able to capture pieces I will want squares which have pieces of the opposite colour to be valid moves.

The function I wrote earlier, `checkSquareForPiece`, returns true if it finds a piece in the given square, or false if otherwise. This is no longer enough information; I need to know if the piece is white or black (i.e. is it capturable). To account for this, I can change the return type of the function to be an int, and return 0 for no piece, 1 for a capturable piece, and 2 for a non-capturable piece. Kings are an exception; they are never capturable, so if the piece is a King 2 will be returned.

```
public static int checkSquareForPiece(Square s) {
    foreach (Piece p in pieces) {
        if (p.square == s) {
            if (p.type == PieceType.KING) return 2;
            if (state.HasFlag(GameState.COLOUR) == p.colour.HasFlag(PieceColour.WHITE))
                return 1;
            else return 2;
        }
    }
    return 0;
}
```

This cannot be tested straight away, since there are now errors elsewhere. In the movement calculation section, I used this function as a boolean value, but now it is an int, so I will need to update to account for this.

### *Pawn*

```
case (PieceType.PAWN): {
    var direction = colour == PieceColour.WHITE ? square.indexY + 1 : square.indexY - 1;
    Square attempt = Chess.GameContainer.findSquareByIndex(square.indexX, direction);
    if (!Chess.checkSquareForPiece(attempt)) moves.Add(attempt);
    return moves;
}
```

For the pawn there is a caveat, they cannot capture forwards, only diagonally. This means we need to add another check. For the original check, we will only add the move to the list if the function returns 0 (no piece). Using a for loop, I can check both the forward diagonals. These should only be added to the list if the function returns 1 (there is a capturable piece).

```

case (PieceType.PAWN): {
    int direction = colour == PieceColour.WHITE ? square.indexY + 1 : square.indexY - 1;
    Square attempt = Chess.GameContainer.findSquareByIndex(square.indexX, direction);
    if (Chess.checkSquareForPiece(attempt) == 0) moves.Add(attempt);
    for (int i = -1; i <= 1; i += 2) {
        attempt = Chess.GameContainer.findSquareByIndex(square.indexX + i, direction);
        if (Chess.checkSquareForPiece(attempt) == 1) moves.Add(attempt);
    }
    return moves;
}

```

This code can actually be simplified into fewer lines. When I am adding 0 to indexX, I will add the move if the result is 0. If I am adding or subtracting 1, I will add the move if the result is 1.

```

case (PieceType.PAWN): {
    int direction = colour == PieceColour.WHITE ? square.indexY + 1 : square.indexY - 1;
    for (int i = -1; i <= 1; i++) {
        Square attempt = Chess.GameContainer.findSquareByIndex(square.indexX + i, direction);
        if (Chess.checkSquareForPiece(attempt) == Math.Abs(i)) moves.Add(attempt);
    }
    return moves;
}

```

### *Bishop*

```

case (PieceType.BISHOP): {
    bool[] tracker = { false, false, false, false };
    int[][] direction = type == PieceType.BISHOP ?
        new int[][] { new int[] { -1, -1 }, new int[] { -1, 1 }, new int[] { 1, -1 }, new int[] { 1, 1 } } :
        new int[][] { new int[] { -1, 0 }, new int[] { 1, 0 }, new int[] { 0, -1 }, new int[] { 0, 1 } };
    int max = Chess.findLargest(new int[] {
        square.indexX - Chess.bounds[0], square.indexY - Chess.bounds[2],
        Chess.bounds[1] - square.indexX, Chess.bounds[3] - square.indexY });
    for (int i = 1; i <= max; i++) {
        for (int j = 0; j < 4; j++) {
            if (!tracker[j]) {
                Square attempt = Chess.GameContainer.findSquareByIndex(
                    square.indexX - i*direction[j][0], square.indexY - i*direction[j][1]) ?? square;
                tracker[j] = Chess.checkSquareForPiece(attempt);
                if (!tracker[j]) moves.Add(attempt);
            }
        }
    }
    return moves;
}

```

tracker will need to be updated to not be a bool. Instead, it will start off as an array of zeroes, and then at each pass through the first for loop, will be set to the value of the result of the function. Then, if the tracker value is 0, the move is added. If it is 2, the move is not added. If it is 1, the move is added, and then the tracker value is set to 2. The initial if statement in the loop will then be changed to if(tracker[j] != 2).

```

case (PieceType.BISHOP): {
    int[] tracker = { 0, 0, 0, 0 };
    int[][] direction = type == PieceType.BISHOP ?
        new int[][] { new int[]{-1,-1}, new int[]{-1,1}, new int[]{1,-1}, new int[]{1,1} } :
        new int[][] { new int[]{-1,0}, new int[]{1,0}, new int[]{0,-1}, new int[]{0,1} };
    int max = Chess.findLargest(new int[] {
        square.indexX-Chess.bounds[0], square.indexY-Chess.bounds[2],
        Chess.bounds[1]-square.indexX, Chess.bounds[3]-square.indexY });
    for (int i = 1; i <= max; i++) {
        for (int j = 0; j < 4; j++) {
            if (tracker[j] != 2) {
                Square attempt = Chess.GameContainer.findSquareByIndex(
                    square.indexX - i*direction[j][0], square.indexY - i*direction[j][1]) ?? square;
                tracker[j] = Chess.checkSquareForPiece(attempt);
                if (tracker[j] != 2) {
                    moves.Add(attempt);
                    if (tracker[j] == 1) tracker[j] = 2;
                }
            }
        }
    }
    return moves;
}

```

### King

```

case (PieceType.KING): {
    string[] attempts = File.ReadAllLines($"res/movement/{type.ToString()}.txt");
    foreach (string att in attempts) {
        Square s = Chess.GameContainer.findSquareByIndex(
            square.indexX + int.Parse(att.Split(',')[0]),
            square.indexY + int.Parse(att.Split(',')[1]));
        if (!Chess.checkSquareForPiece(s) && s != null) moves.Add(s);
    }
    return moves;
}

```

This will just be a simpler version of the pawn fix. All that needs to be done is change the last if statement to `if (Chess.checkSquareForPiece(s) != 2 && s != null)`:

```

case (PieceType.KING): {
    string[] attempts = File.ReadAllLines($"res/movement/{type.ToString()}.txt");
    foreach (string att in attempts) {
        Square s = Chess.GameContainer.findSquareByIndex(
            square.indexX + int.Parse(att.Split(',')[0]),
            square.indexY + int.Parse(att.Split(',')[1]));
        if (Chess.checkSquareForPiece(s) != 2 && s != null) moves.Add(s);
    }
    return moves;
}

```

All other pieces call the code of one of the pieces already covered, so they are also fixed. I can now do some testing for these new calculations since all errors are fixed.

### *Pawn*

- \* The white pawn will be clicked.
- \* The upper right square should become highlighted. The other two should be blocked by the knight and bishop.
- \* The outcome was as expected.

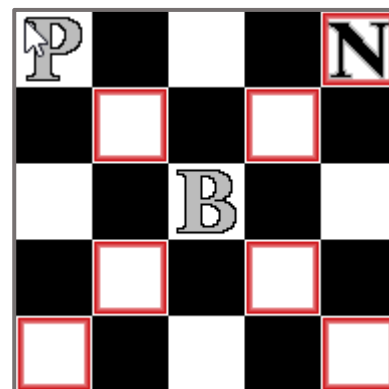
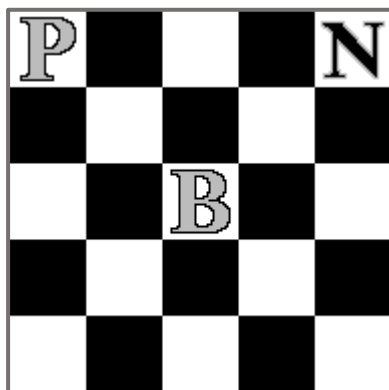


- \* The black pawn will be clicked.
- \* The square below the pawn should become highlighted.
- \* The outcome was as expected.



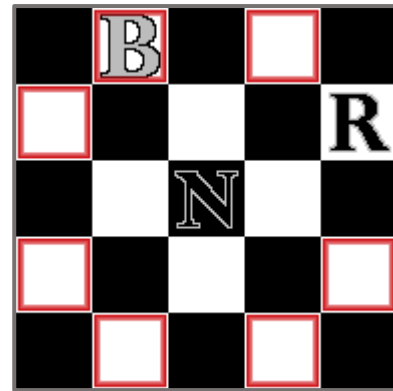
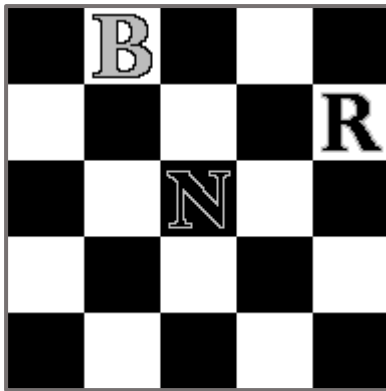
### *Bishop*

- \* The white bishop will be clicked.
- \* The black knight should become highlighted, and the white pawn should not.
- \* The outcome was as expected.



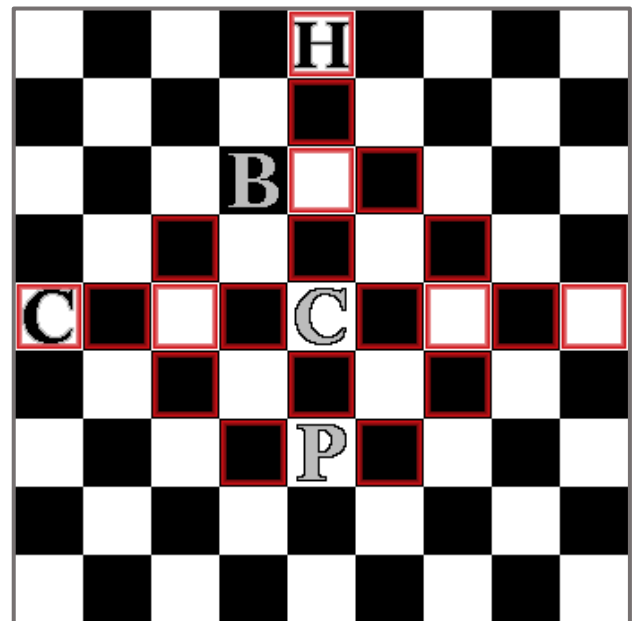
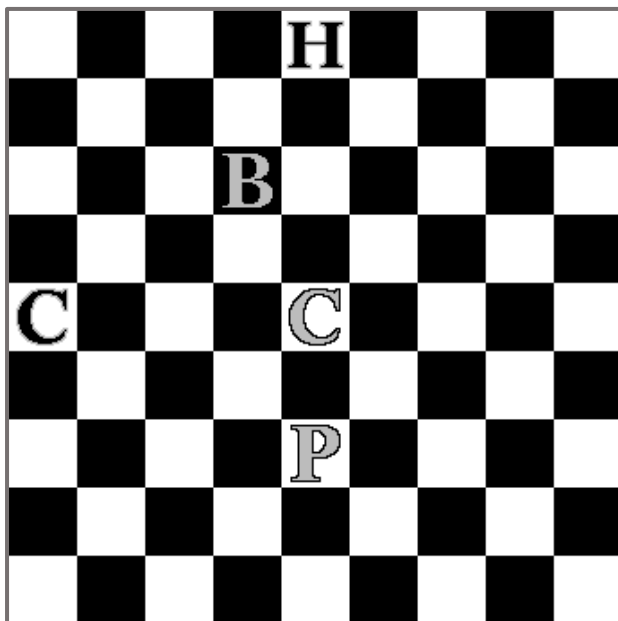
### *Knight*

- \* The black knight will be clicked.
- \* The white bishop should become highlighted, and the black rook should not.
- \* The outcome was as expected.



### *Chancellor*

- \* The white chancellor will be clicked.
- \* The upper line should include the black hawk and then stop. The left line should include the black chancellor and then stop.
- \* The lower line should be blocked by the white pawn.
- \* The white bishop should not become highlighted.
- \* The outcome was as expected.

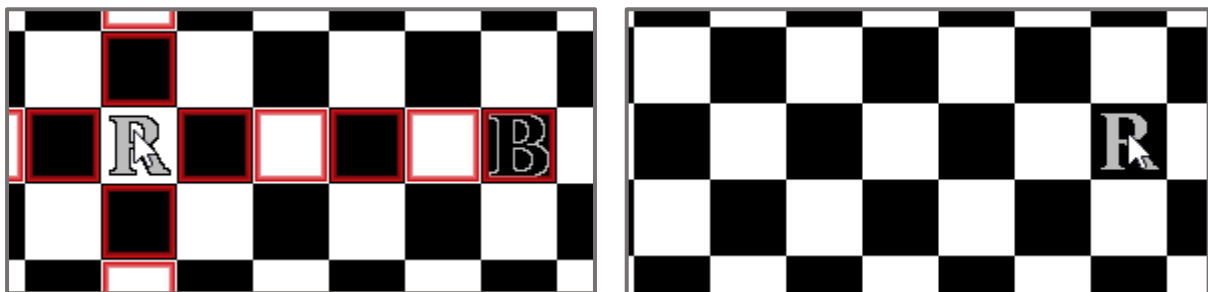


I can conclude that piece movement calculations correctly take into account the possibility of capturing enemy pieces.

Currently, even though I can move a piece onto the same square as another, that isn't actually capturing. I need to remove a piece from the game once it gets captured. The most sensible place to implement this is in the `move` function. When a move is selected, the game will check if the selected move has a piece on it already. If there is a piece there, it will be removed.

```
public void move(Square s) {
    foreach (Piece p in Chess.pieces) {
        if (p.square == s) { Chess.pieces.Remove(p); break; }
    }
    square = s;
}
```

Testing it out:



It looks to be working fine.

## Check and Checkmate

To put the game into a state where it is playable, there needs to be a concept of check and checkmate.

After every turn, some function will be run which checks the movement of each piece, and if one of them contains the king of the opposite colour, the check flag will be set, meaning on the next turn only the king can be moved, and it must be moved in a way which puts it out of check.

Currently there is an issue, the calculate movement function does not include kings of the opposite colour, because these should not be capturable. However, if I changed the function to include them, they would be shown as capturable to the player. What I need is one function for the player, and one for the check function.

To do this, I can make `calculateMovement` take a parameter, `includeKings`, which will determine whether kings will be included. The best way to make this work will be to add another parameter to `checkSquareForPiece` which will decide if the condition which returns 2 if the piece is a king is active.

The updated checkSquareForPiece looks like this:

```
public static int checkSquareForPiece(Square s, bool includeKings) {
    foreach (Piece p in pieces) {
        if (p.square == s) {
            if (p.type == PieceType.KING && !includeKings) return 2;
            if (state.HasFlag(GameState.COLOUR) == p.colour.HasFlag(PieceColour.WHITE))
                return 1;
            else return 2;
        }
    }
    return 0;
}
```

And the beginning of the updated movement calculation function:

```
public List<Square> calculateMovement(bool includeKings) {
    List<Square> moves = Square.emptyList();
    switch (type) {
        case (PieceType.PAWN): {
            int direction = colour == PieceColour.WHITE ? square.indexY + 1 : square.indexY - 1;
            for (int i = -1; i <= 1; i++) {
                Square attempt = Chess.GameContainer.findSquareByIndex(square.indexX + i, direction);
                if (Chess.checkSquareForPiece(attempt, includeKings) == Math.Abs(i)) moves.Add(attempt);
            }
            return moves;
        }
    }
}
```

The only thing that has changed is the addition of the extra parameter throughout the function.

The first iteration of evaluateCheck function looks like this:

```
public void evaluateCheck() {
    var colour = state.HasFlag(GameState.COLOUR) ? PieceColour.BLACK : PieceColour.WHITE;
    Square king = pieces.Find(p => p.type == PieceType.KING && p.colour == colour).square;

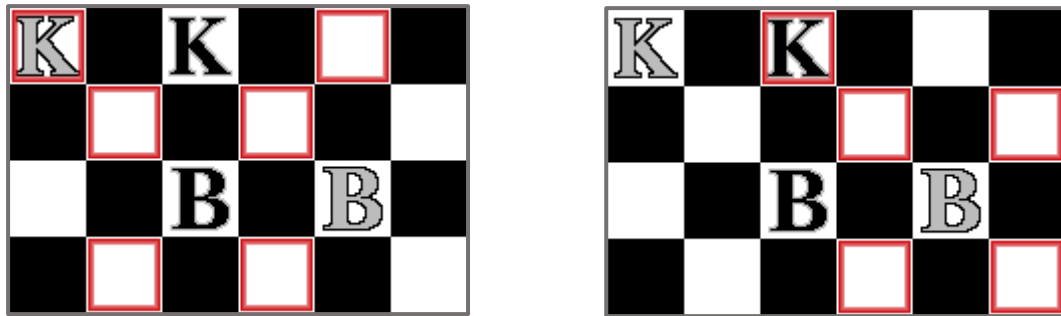
    foreach (Piece p in pieces) {
        if (p.calculateMovement(true).Contains(king)) {
            state ^= GameState.CHECK; break;
        }
    }
}
```

This function is called every time a piece moves. I also have a debug label set up to display the game state every time handleTurn is called. I can use this to see if this function is working correctly. During normal gameplay, state alternates between 0 and 1 (representing white and black's turn, respectively). Check, being the third flag in GameState, adds 4 to state if it is set. This means that if white is put into check, I would expect to see a value of 4, and for black a value of 5.

I test this by creating one king of each colour and a bishop of each colour. However, the label only ever has a value of 1 or 0, even if the pieces move to a location which would put one of the kings into check.



I'm not sure where the problem is, so I will be being by making sure the updated calculateMovement function is working. To do this, I will simply adjust drawMoves to draw the result of calculateMovement(true) instead of calculateMovement(false).



The kings are being included, as expected, so it seems this is not the issue.

The next thing to try is if king, the square which will contain the king of the opposite colour, is valid and correct. To do this I can set up the debug label to output the coordinates of this square and compare it using the other debug label which will output the square the cursor is currently over. However, this does not seem to be the issue either, as after moving a white piece and hovering over the black king, both labels have the same value.

```
7, 12, 266, 114
7, 12, 266, 114
```

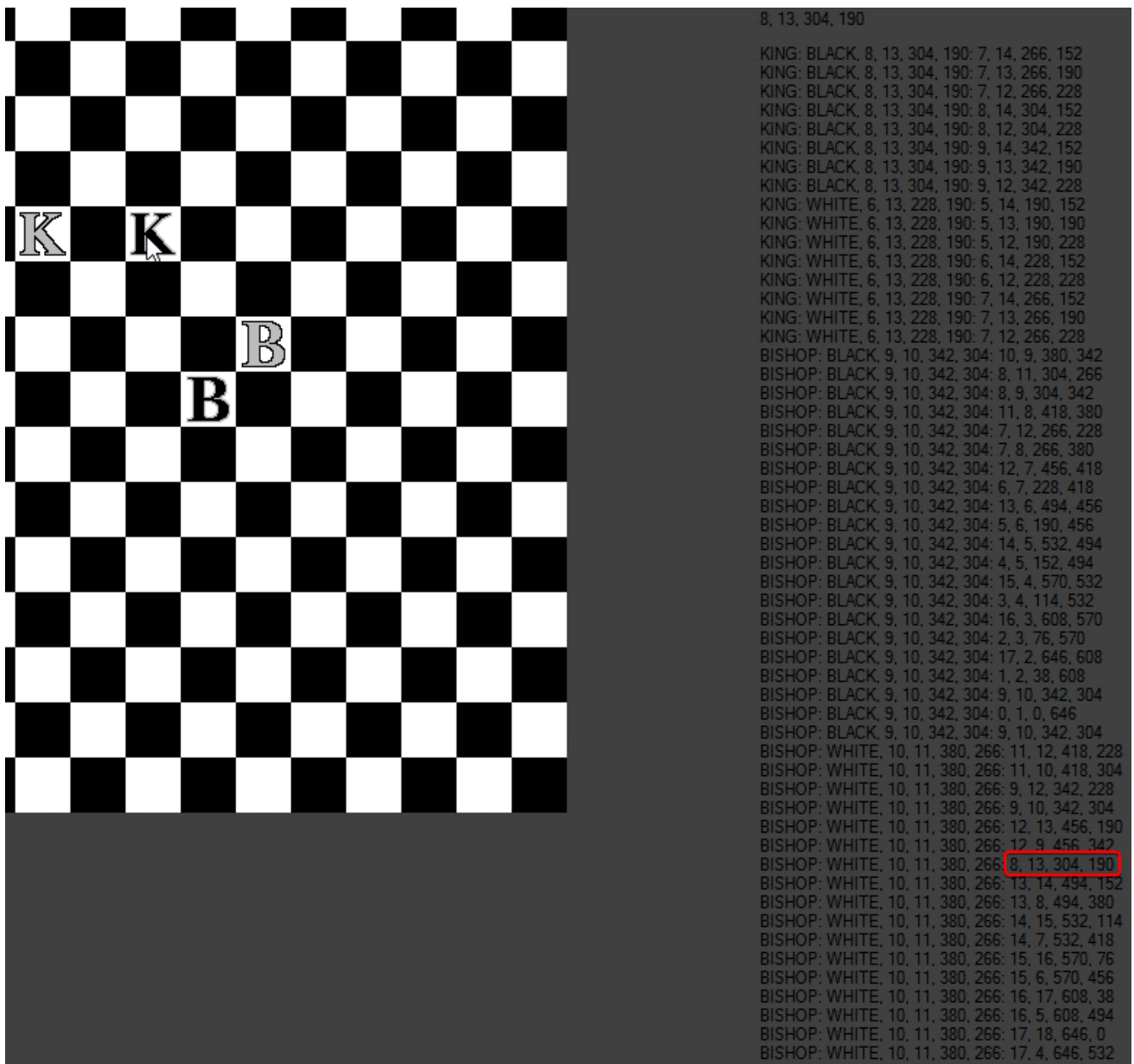
Next, I will check if my condition for setting the state is correct. I will do this by adding the following line of code into the foreach loop:

```
c.debug3.Text += p.calculateMovement(true).Contains(king).ToString();
```

This will tell me the value of this condition for each piece on the board. If it is working correctly, I would expect one of these to be true. However:

```
FalseFalseFalseFalse
```

I would have expected to find the problem by now. Since I haven't, I will need to display more information. I decide that the next thing to do should be to list each Square contained in the movement for each piece.



The black king is at [8,13]. The bishop is in a position to put this king in check, so its movement should include it. As can be seen, the highlighted value shows the square that the king is on, meaning the list of available movement does contain the square the king is on, so there is a problem with the method I have used to decide whether a list contains a square.

After doing some research, `.Contains` uses the default equality check for the specified object type. I have not defined one specifically for `Square`, and I don't know the exact method it uses to compare otherwise, so I will try overriding this method to see if that fixes the problem.

This is done using the following:

```
public bool Equals(Square s) => (X == s?.X && Y == s?.Y && indexX == s?.indexX && indexY == s?.indexY);
public override bool Equals(object obj) {
    if (obj == null) return false;
    Square s = obj as Square;
    if (s == null) return false;
    else return Equals(s);
}
public override int GetHashCode() => X.GetHashCode() ^ Y.GetHashCode() ^ indexX.GetHashCode() ^ indexY.GetHashCode();
```

I have to write three functions. The first compares two Squares with each other, the second compares an object with an instance of Square (this is the method used by `.Contains`, this method also uses the first method), and the third generates the hash code for an instance of Square. I do not actually need this, but it is required that this method is overrode if `Equals` is, and vice versa.

Unfortunately, this does nothing. My last resort is to set a breakpoint in `evaluateCheck` and step through the code line-by-line to see what is happening, using the watch feature to see the values of important variables:

▷ this	{InfiniteChess.Chess+GameConta
▷ p	{KING: WHITE, 11, 14, 418, 38}
▷ s	{10, 15, 380, 0}
▷ king	{13, 14, 494, 38}
▷ a	Count = 8

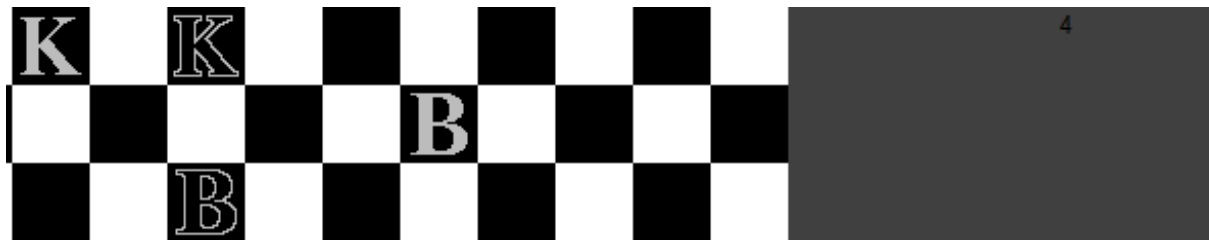
After a lot of pressing the next line button, I finally find the problem. In `checkSquareForPiece`, I check the colour of the piece being processed against the colour of the current turn (stores in `state`). This was assuming that the function is called during a turn (i.e. before the colour is updated in `state`). However, `evaluateCheck` is called after the `state` has already been updated. This was causing the movement function to not be adding the square with the opposite colour king (instead it was adding the same colour king), and therefore the game could never enter check. To fix this, I move `evaluateCheck` to be before the updating of `state`, and swap around the colour of king to look for in relation to the current `state`. The new version looks like this:

```
public void evaluateCheck() {
    var colour = state.HasFlag(GameState.COLOUR) ? PieceColour.WHITE : PieceColour.BLACK;
    Square king = pieces.Find(p => p.type == PieceType.KING && p.colour == colour).square;
    foreach (Piece p in pieces) {
        if (p.calculateMovement(true).Contains(king) && p.colour != colour) {
            state ^= GameState.CHECK; break;
        }
    }
}
```

Testing this:



It is black's turn (state contains 1), and black is in check because of the white bishop (state contains 4), therefore the state should be  $4+1=5$ , which can be seen on the right.



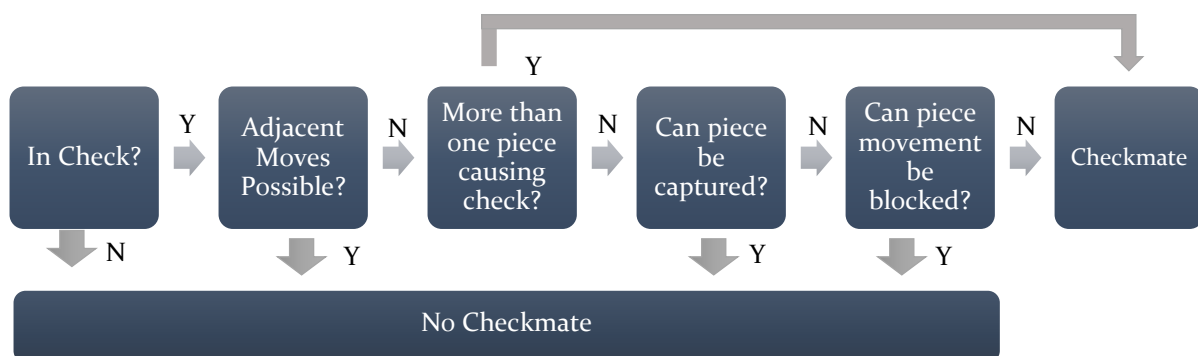
Similarly, it is white's turn (0) and is in check because of the black bishop (4), so the state should be  $0+4=4$ , which is what it actually is.

Moving out of check does not actually update the state, since the state is only updated when the check condition is met (in which case the state is flipped). To fix this, I will just remove the check flag from the state if it is enabled every time `evaluateCheck` is called:

```
if (state.HasFlag(GameState.CHECK)) { state ^= GameState.CHECK; }
```

A side effect of this is that if one player is put into check, they will be automatically removed from it as far as the state is concerned after their turn is over. Currently, this leads to some strange behaviour, but one of the rules of chess is that if you are in check then you have to make a move which will put you out of check. This means that there is no need to deal with this side effect right now, as it will be intended behaviour soon enough.

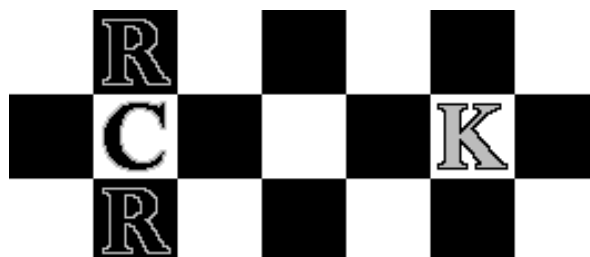
Dealing with checkmate will be a little more challenging. If one player is in checkmate, it means that they are in check, but also there are no available squares for the king to move, and no other piece can make a move which will remove the king from check (by capturing or blocking). The following diagram represents the process I will use to determine checkmate:



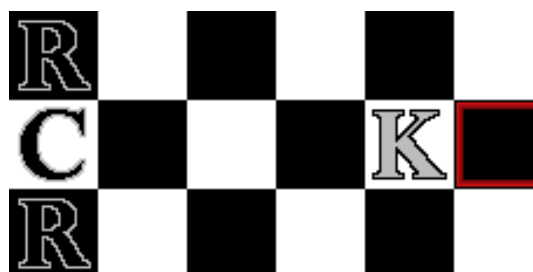
I have just completed the ability to determine whether a player is in check, so the next step is to be able to check if any of the squares adjacent to the king can be moved to. If there are squares that the king can move to that would not put it in check, then it cannot be in checkmate. I can add this as an extension of `calculatePieceMovement` for only the king pieces:

```
case (PieceType.KING): {
    string[] attempts = File.ReadAllLines($"res/movement/{type.ToString()}.txt");
    foreach (string att in attempts) {
        Square s = Chess.GameContainer.findSquareByIndex(
            square.indexX + int.Parse(att.Split(',')[0]),
            square.indexY + int.Parse(att.Split(',')[1]));
        if (Chess.checkSquareForPiece(s, includeKings) != 2 && s != null) moves.Add(s);
    }
    if (type != PieceType.KING) return moves;
    for (int i = -1; i < 2; i++) {
        for (int j = -1; j < 2; j += (2 - Math.Abs(i))) {
            foreach (Piece p in Chess.pieces) {
                Square attempt = Chess.GameContainer.findSquareByIndex(square.indexX + i, square.indexY + j);
                if (p.type != PieceType.KING & p.colour != colour) {
                    if (p.calculateMovement(false).Contains(attempt))
                        moves.Remove(attempt);
                }
            }
        }
    }
    return moves;
}
```

This will check if there are any pieces of the opposite colour which can move to one of the squares around the king, and removes it from the list of available moves if necessary. This works in most cases, but there are some cases where this logic does not apply. For example, consider the following scenario:



This is a checkmate (assuming there are no other pieces on the board), there is nowhere the king can move to avoid check. However, if we try to move the king, the algorithm above returns the following:

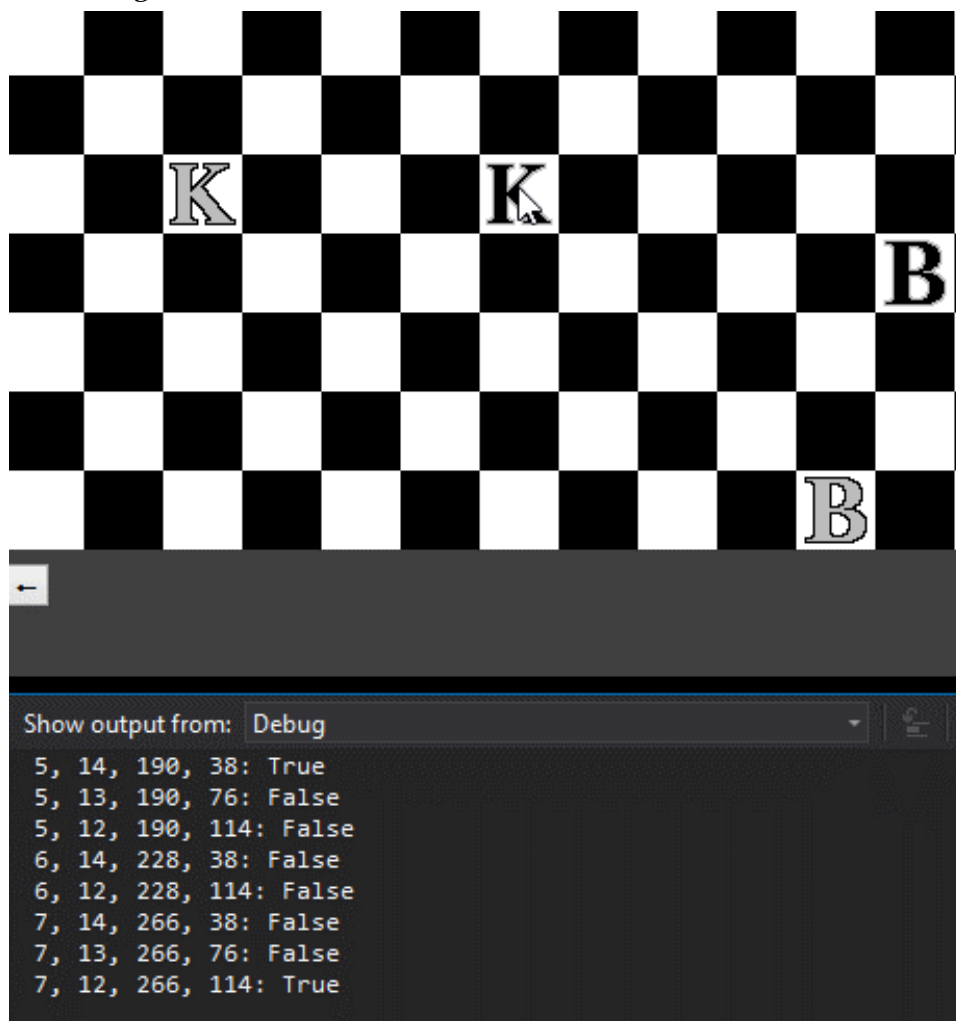


This is because when the movement of the king is being evaluated, that square is not part of the available movement of the chancellor, because it is being blocked by the king. However, once the king moves there, it is no longer being blocked. This means I will need a different approach to the problem.

A different solution would be to iterate through each possible move for the king and evaluate check for this new board. To check if it is working correctly initially, I will use `Debug.WriteLine` to write each square tested and the outcome of the check for that square when checkmate is evaluated. The first iteration of the function looks like this:

```
public void evaluateCheckMate(PieceColour c) {
    Piece king = pieces.Find(p => p.type == PieceType.KING && p.colour == c);
    Square original = king.square;
    foreach (Square s in king.calculateMovement(false)) {
        king.move(s);
        var a = evaluateCheck(c);
        Debug.WriteLine($"{s.ToString()}: {a}");
    }
    king.move(original);
}
```

The black king is at [6,13], and the bishop was moved to put it into check. Given this arrangement, I would expect [7,12] and [5,14] to return true (moving to these squares would put the king in check), and the others to return false.



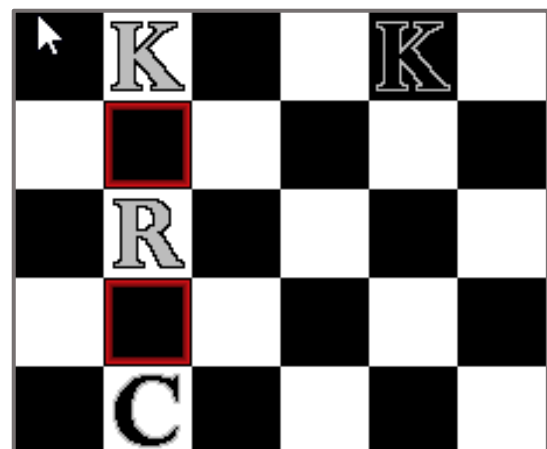
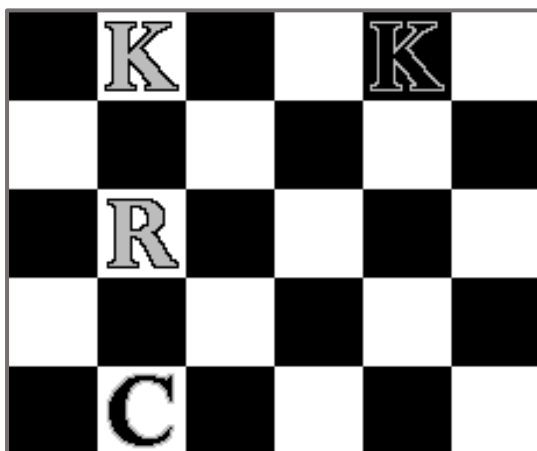
This was the output, which is in line with the expectations. However, in future, I will want to be able to draw these to the board, because the output of this function will represent the actual available movement for the king (as opposed to the

calculateMovement function, which does not consider check). Not only this, but I will also need to consider a similar case for other pieces; if a piece is blocking the line between an enemy piece and a king, then this piece cannot move out of the way so as to cause check. To solve this, I need to add some form of check for check in the movement calculation function. However, the current evaluateCheck contains calculateMovement calls in it already. A way around this would be to have a 'primitive' movement calculation which does not take check into account, and then another function which will take the result from that function, try every move, and then remove any which would cause check. Combined, these should give valid movement for all pieces, taking the possibility of check into account.

I have renamed the previous movement function to calculateInitMovement, and then added a new calculateMovement as follows:

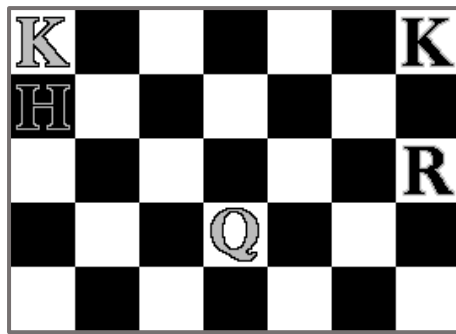
```
public List<Square> calculateMovement(bool includeKings) {
    List<Square> moves = calculateInitMovement(includeKings);
    List<Square> newMoves = new List<Square>();
    Square original = square;
    foreach (Square s in moves) {
        newMoves.Add(s);
        move(s);
        Square king = Chess.pieces.Find(p => p.type == PieceType.KING && p.colour == colour).square;
        foreach (Piece p in Chess.pieces) {
            if (p.colour == colour) continue;
            if (p.calculateInitMovement(true).Contains(king)) { newMoves.Remove(s); }
        }
        move(original);
    }
    return newMoves;
}
```

To test it out, I have the scenario on the left. The rook is blocking the white king from being in check via the black chancellor. If the rook were to move to either side, the king would be in check. This means that moving to either side should not be a valid move. Clicking the white rook yields the following moves, which are indicative that this code works as expected.



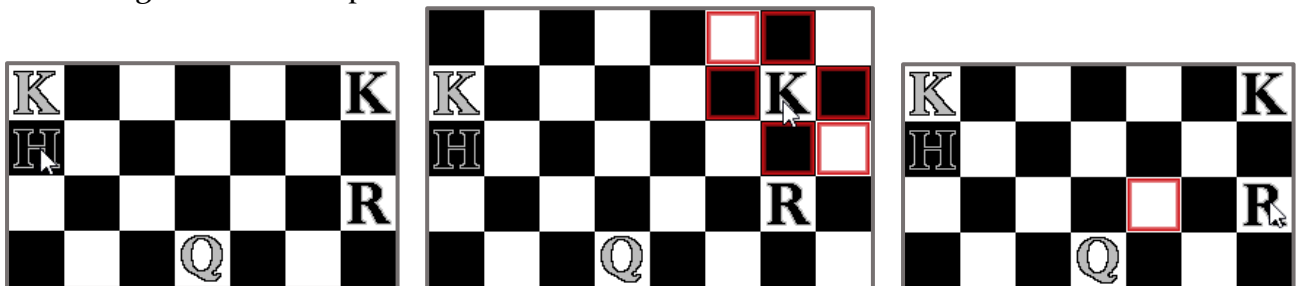


On the left is another configuration. Black is in check by the white queen, and it is



black's turn (white is not in check; hawks move two or three spaces in any direction, but not one). Since black is in check, the only legal moves are ones that will take black out of check. For the hawk, there are no moves that will remove check, so there should be no moves shown. The rook could move two squares to the left to block the path of the queen, but

nowhere else. The king itself could move anywhere except on the line formed by it and the queen to get out of check. The results of clicking each of these pieces can be seen below.



However, currently as a part of the move trialling process, piece capturing is not considered. This is because I have to undo each move I try to evaluate check, but I have not currently implemented any way to undo a piece capture, so this will be the next thing I will do. Later on, I will implement a full move history with undoing of any number of moves, but right now I just need to undo by one move.

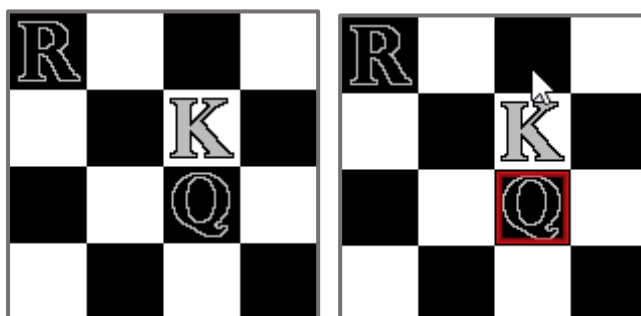
There will be a variable called `lastMove` which is part of the `Chess` class. This will have a default value of null, but whenever a piece is captured, it will hold a copy of the piece

```
public void move(Square s) {
    Piece p = Chess.pieces.Find(q => q.square == s);
    if (p != null) {
        Chess.pieces.Remove(p);
        Chess.lastMove = p;
    }
    square = s;
}
```

that was captured, so that it can be put back into the game if the move is undone. I will then update `Piece.move` to use this value appropriately, and `calculateMovement` to be able to reset this value if it was used.

```
move(original);
if (Chess.lastMove != null) { Chess.pieces.Add(Chess.lastMove); Chess.lastMove = null; }
```

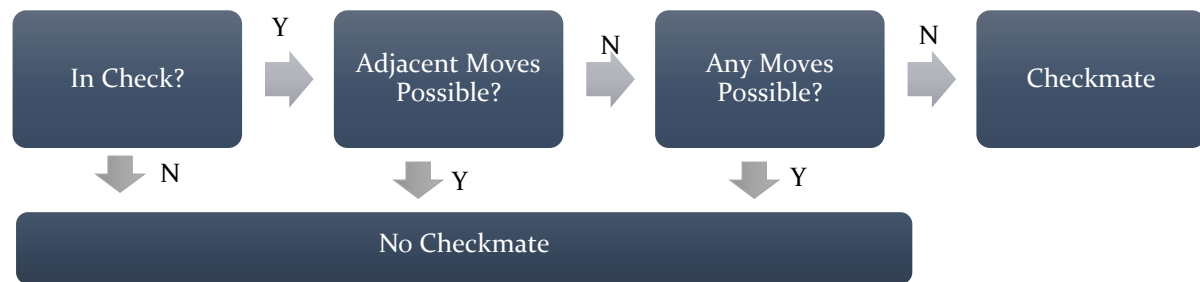
On the left is a scenario. With the previous code, there would be no viable moves for the king. With the new code, the expected result is that capturing the queen is an



available move, which would remove the king from check. As can be seen, this is exactly what happens, which indicates the code works correctly.



With this new infrastructure for evaluating check and calculating moves, checking for checkmate is now a lot simpler. The new process can be represented by this diagram:

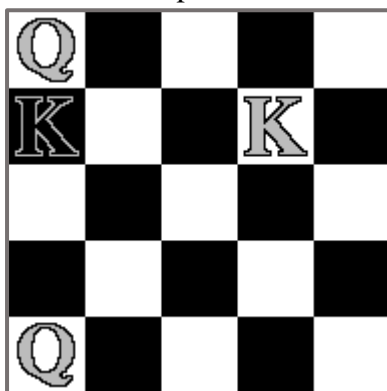


This means evaluateCheck will look like this:

```

public bool evaluateCheckMate(PieceColour c) {
    Piece king = pieces.Find(p => p.type == PieceType.KING && p.colour == c);
    if (king.calculateMovement(false).Count() != 0) return false;
    var playerPieces = from p in pieces where p.colour == c select p;
    foreach (Piece p in playerPieces) { if (p.calculateMovement(false).Count() != 0) return false; }
    return true;
}
  
```

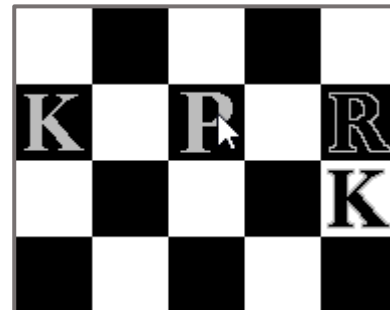
To the left is an initial configuration. It is white's turn, and the queen below the black king moves to the position indicated in the image on the right. This is of course checkmate. This means the WIN flag (0x08) should be set for the player that has won after this turn is complete. Looking to the right, the game state is 12, which is WIN 0x08 + CHECK 0x04. The lack of COLOUR 0x01 signifies this is referring to white. Therefore, state 12 represents white having won the game, which is correct.



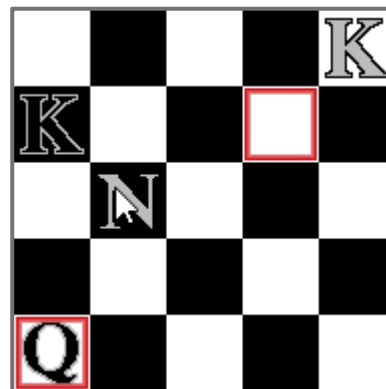
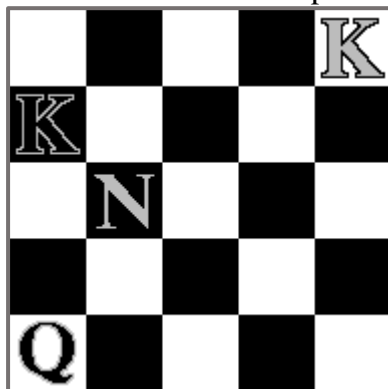
I will now do some movement tests with the new calculations to ensure it is working properly. If they are successful, this would mean the game is playable as a chess game with two human players.

### Updated Piece Movement Tests

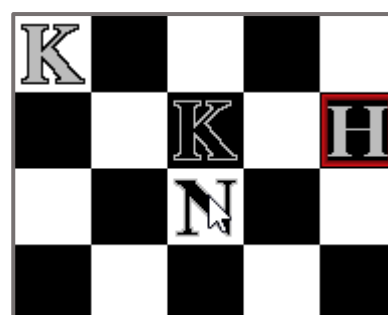
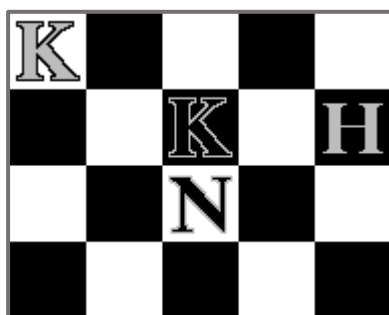
- \* The white pawn will be clicked.
- \* No squares should become highlighted. Any move by the pawn would put the white king in check.
- \* The outcome was as expected.



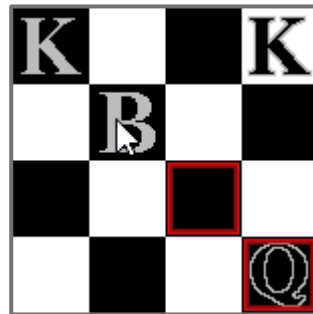
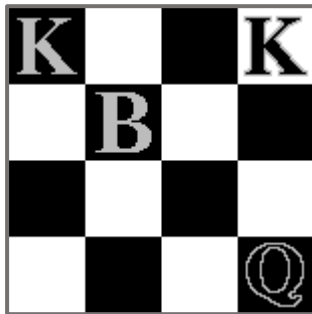
- \* The white knight will be clicked.
- \* The black queen and the square to the bottom left of the white king should become highlighted. Other moves would not remove the white king from check.
- \* The outcome was as expected.



- \* The black knight will be clicked.
- \* The white hawk should become highlighted. All other moves would not remove the black king from check.
- \* The outcome was as expected.



- \* The white bishop will be clicked.
- \* The two squares to the lower right should become highlighted. The other squares would put the white king in check.
- \* The outcome was as expected.

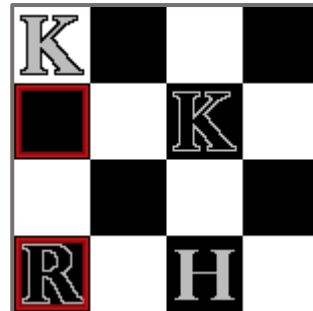
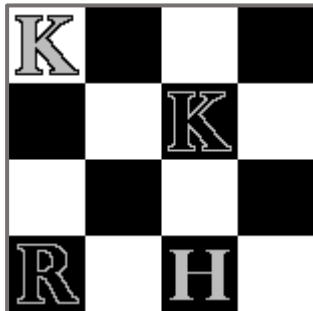


- \* The black rook will be clicked.
- \* One square to the right and two squares to the left should become highlighted. The other squares would put the black king in check.

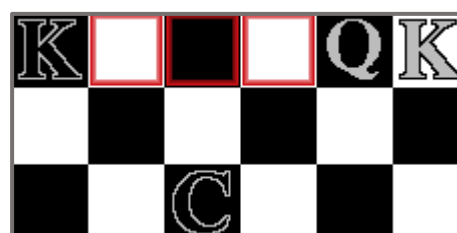
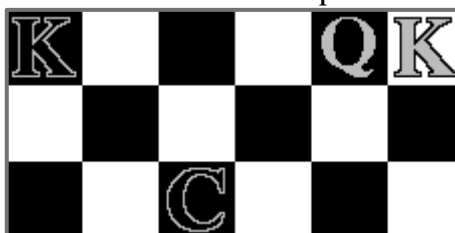


- \* The outcome was as expected.

- \* The white hawk will be clicked.
- \* The rook and the square below the white king should become highlighted.
- \* The outcome was as expected.



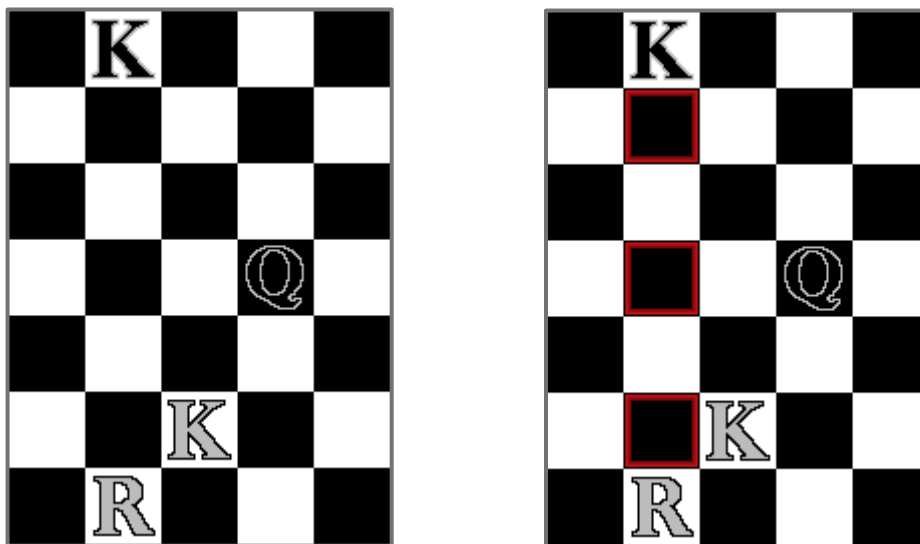
- \* The black chancellor will be clicked.
- \* The three squares between the black king and the white queen should become highlighted.
- \* The outcome was as expected.



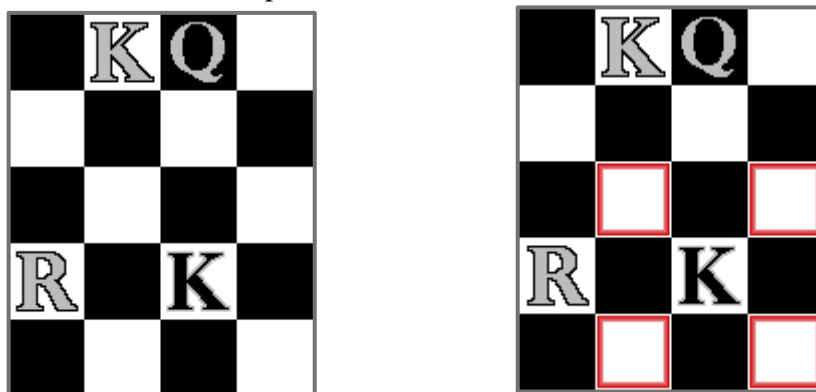
- \* The white mann will be clicked.
- \* The black queen and the square to the right of it should become highlighted.
- \* The outcome was as expected.



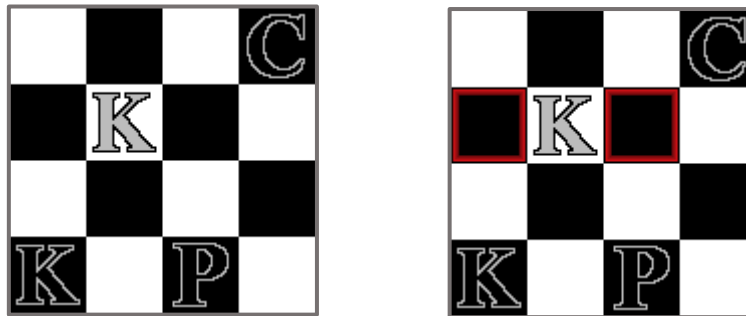
- \* The black queen will be clicked.
- \* The square two to the left of the queen, and the two squares which two squares above and below this square should become highlighted.
- \* The outcome was as expected.



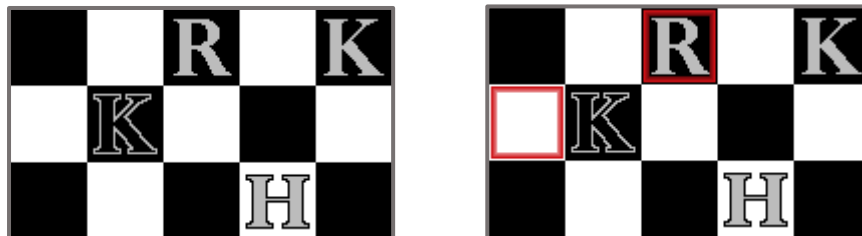
- \* The black king will be clicked.
- \* The 4 diagonal squares around the king should become highlighted. The others would put the king in check.
- \* The outcome was as expected.



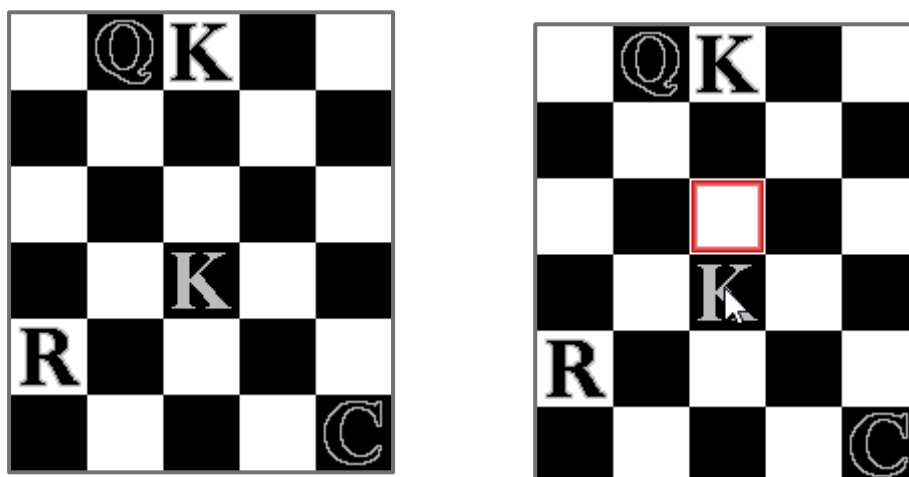
- \* The white king will be clicked.
- \* The squares to the left and right should become highlighted.
- \* The outcome was as expected.



- \* The black king will be clicked.
- \* The squares to the left and top right should become highlighted.
- \* The outcome was as expected



- \* The white king will be clicked.
- \* The square above the white king should become highlighted.
- \* The outcome was as expected.



I can conclude from these results that movement works when taking the possibility of check into account. This means the game is now in a state where it can be played fully by two human players.

## Move History

The main feature that isn't directly gameplay for the game will be the move history. As mentioned in the design section, this is a list which will describe all moves that have happened in the game, written in the standard notation for infinite chess. As moves happen, they will be added to the list. I will want to be able to manage and edit the data of stored moves in a convenient way, so I will make `MoveHistory` a class which extends `TextBox`.

`MoveHistory` will store a list which contains each move in string form. It will also have two methods responsible for adding and removing moves from the list. The framework for this class looks like this:

```
public class MoveHistory : TextBox
{
    public List<string> moves { get; private set; }

    public void addMove(string m) {
        moves.Add(m);
        Lines = moves.ToArray();
    }

    public string undoMove() {
        string a = moves.Last();
        moves.Remove(a);
        Lines = moves.ToArray();
        return a;
    }
}
```

The way I want this to work is that in `handleTurn`, instead of calling `Piece.move` directly, I will call `history.addMove`, which will both call `Piece.move` and handle updating the history display. An important point is that `calculateMovement` will continue to use `Piece.move` as opposed to `history.addMove` because I do not want every possible move being added to the history every time a player clicks on a piece.

`addMove` will need to take in the piece that is moving and the square it is moving to as arguments. It will then put the information needed for the history (first letter of piece type, square it's on etc.) into a string. The piece will be moved, and the string to be written to the history will be completed based on whether there was a piece in the square that was just moved to. I could use `checkSquareForPiece` to determine if there was a piece in the square that was moved to, but this seems rather unnecessary since `Piece.move` already does this. A better solution is to use an out parameter in `Piece.move`. This means that I can have the option to use a return value from the function, but still keep the type `void` so I am not forced to use the function as a value. The out parameter will return a piece, which will be `null` if there was no piece on the square that was moved to or will contain the information for the piece if there was one.

This new function looks like this:

```
public void move(Square s, out Piece pOut) {
    Piece p = Chess.pieces.Find(q => q.square == s);
    pOut = p;
    if (p != null) {
        Chess.pieces.Remove(p);
        Chess.lastMove = p;
    }
    square = s;
}
```

And I can now use this parameter in the history code like this:

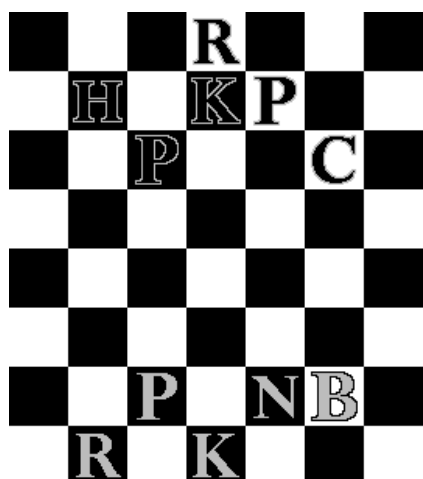
```
public void addMove(Piece p, Square s) {
    string[] data = p.ToString().Split(',');
    string moveText = $"{data[0].Substring(0,1)}({data[2]},{data[3]})";
    p.move(s, out Piece pOut);
    if (pOut != null) {
        data = pOut.ToString().Split(',');
        moveText += $"x{data[0].Substring(0,1)}({data[2]},{data[3]})";
    }
    else {
        moveText += $"-({s.indexX},{s.indexY})";
    }
    moves.Add(moveText);
    Lines = moves.ToArray();
}
```

This code will add moves to the history which will occur. The format these are in is described in the Game Design section, but the basic idea is to print the first letter of the type of piece that moved, then it's starting index, then a x if it captured something and a - if not, followed by the first letter of the piece captured (if there was one), and finally the square that was moved to. Here, I create an array called data which just takes the result of the method p.ToString() and splits it into an array based on where any commas are in the string. p.ToString() returns the type, colour, and square of the piece (using Square.toString(), which in turn returns the indexes and coordinates of a square). These two functions can be seen below.

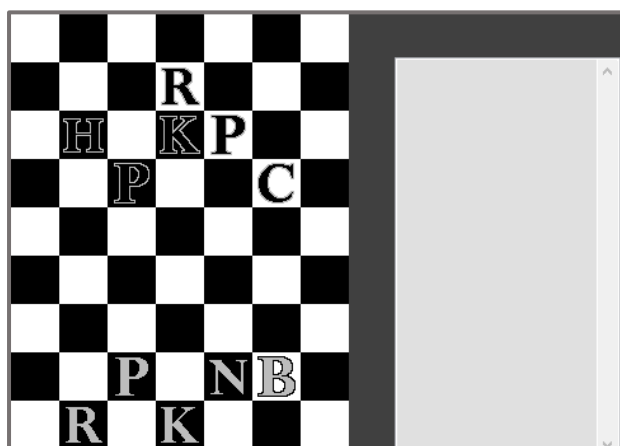
```
public override string ToString() => $"{type},{colour},{square.ToString()}";

public override string ToString() =>
    $"{indexX.ToString()},{indexY.ToString()},{X.ToString()},{Y.ToString()}";
```

Now to test out this iteration, let's start with the scenario seen on the right. White of course goes first. I will then execute the following moves:



1. The white rook captures the black hawk.
2. The black king moves behind the right black pawn.
3. The white rook moves to the black kings previous location.
4. The black chancellor captures the white rook.
5. The white king moves to the right.
6. The black chancellor moves between the white pawn and knight.
7. The white bishop moves forward and to the right.
8. The black rook moves two to the left.
9. The white bishop captures the black rook.



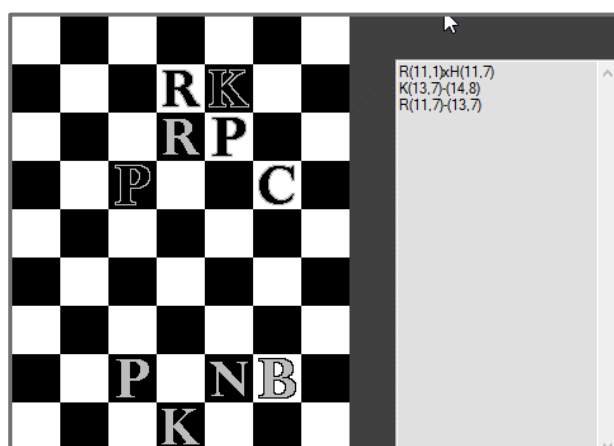
0



1

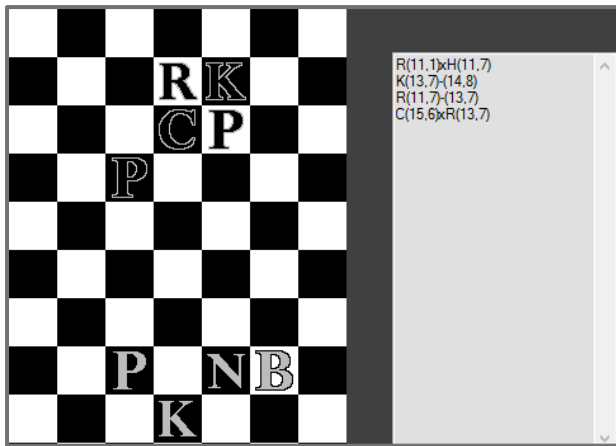


2

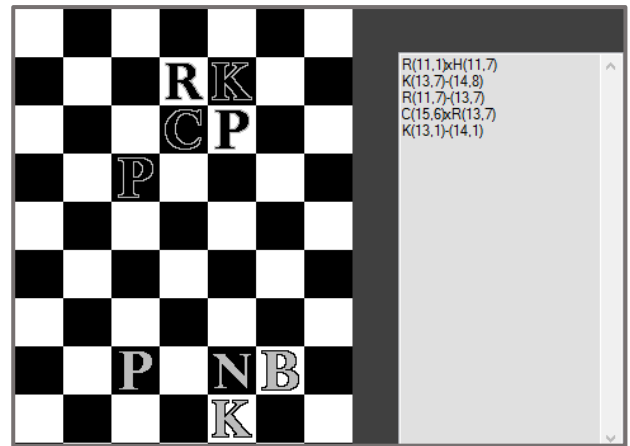


3

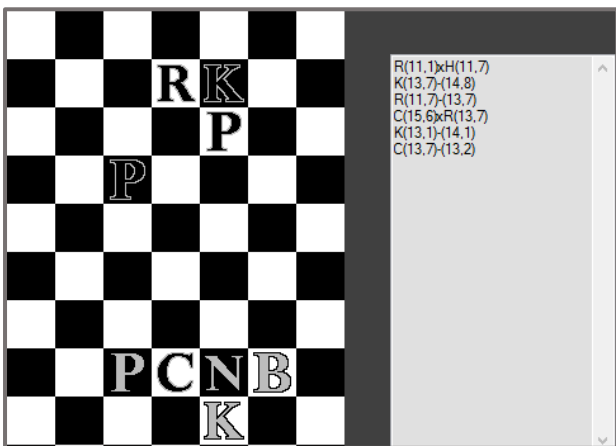




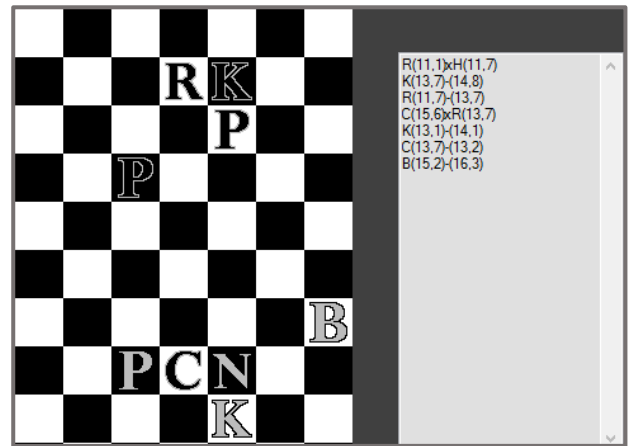
4



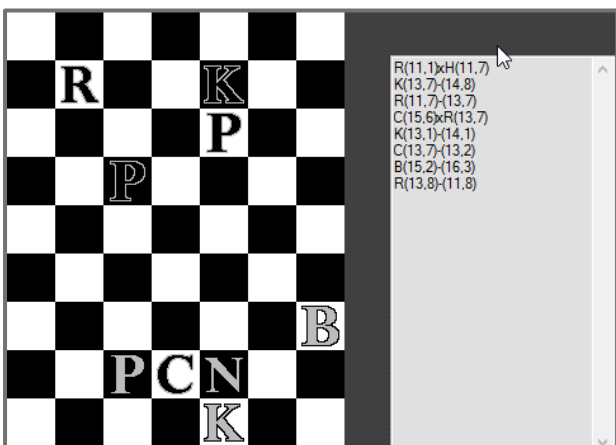
5



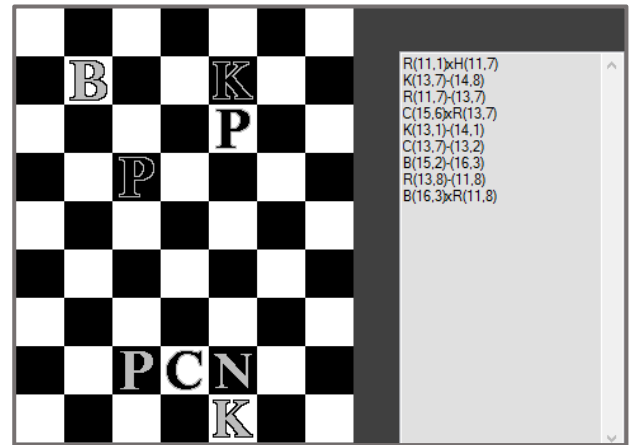
6



7



8



9

To the right in each screenshot is the move history at that point. It would appear that so far it is working correctly.



## Section 4: Evaluation

---

## Section 5: Bibliography

---

### References

<sup>1</sup> <http://www.pathguy.com/chess/ChessVar.htm>

<sup>2</sup> [https://en.wikipedia.org/wiki/Algebraic\\_notation\\_\(chess\)](https://en.wikipedia.org/wiki/Algebraic_notation_(chess))

<sup>3</sup> <http://www.chessvariants.com/invention/chess-on-an-infinite-plane>

### Sources

<sup>A</sup> [https://cdn-images-1.medium.com/max/1600/1\\*UA5VlNs7s4gl8oVknAo99w.jpeg](https://cdn-images-1.medium.com/max/1600/1*UA5VlNs7s4gl8oVknAo99w.jpeg)